Performance Analysis of Dynamic Routing Protocols in a Low Earth Orbit Satellite Data Network

THESIS

Richard F. Janoso
Captain, USAF

AFIT/GE/ENG/96D-08

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/96D-08

Performance Analysis of Dynamic Routing Protocols in a Low Earth Orbit Satellite Data Network

THESIS

Richard F. Janoso
Captain, USAF

AFIT/GE/ENG/96D-08

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U. S. Government.

AFIT/GE/ENG/96D-08

Performance Analysis of Dynamic Routing Protocols in a Low Earth Orbit Satellite Data Network

THESIS

Presented to the faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

Richard F. Janoso, B.S.E.E.

Captain, USAF

December, 1996

Acknowledgments

There are many people who have been of invaluable help in completing this thesis. Specifically, I'd like to thank my thesis advisor, Captain Richard A. Raines, for patient guidance, and my thesis committee Lt. Col. David L. Coulliette and Lt. Col. David M. Gallagher, for their specialized support. I'd also like to thank my classmates, especially Captain Douglas K. Stenger for providing various parts of the simulation system, and the members of the WC2 group (Capts Brian Ernisse, Rod Radcliffe, and Eric Zeek) for downtime recovery.

Richard F. Janoso

Table of Contents

List of Figures

Abstract

Modern warfare is placing an increasing reliance on global communications. Currently under development are several Low Earth Orbit (LEO) satellite systems that propose to deliver voice and data traffic to subscribers anywhere on the globe. However, very little is known about the performance of conventional routing protocols under orbital conditions where the topology changes on a scale of minutes rather than days.

This thesis compares two routing protocols in a LEO environment. One (Extended Bellman-Ford) is a conventional terrestrial routing protocol, while the other (Darting) is a new protocol which has been proposed as suitable for use in LEO networks. These protocols are compared via computer simulation in two of the proposed LEO systems (Globalstar and Iridium), under various traffic intensities. Comparative measures of packet delay, convergence speed, and protocol overhead are made

It was found both protocols were roughly equivalent in end-to-end delay characteristics, though the Darting protocol had a much higher overhead load and demonstrated higher instability at network update periods. For example, while steady-state end-to-end delays were within a few milliseconds, in one case Darting showed an increase of 764% in convergence time over Extended Bellman-Ford with an increase of 149% in overhead. Over all cases, Darting required an average of 72.1% more overhead than Extended Bellman-Ford to perform the same work. Darting was handicapped by its strong correlation between data traffic and protocol overhead. Modifications to reduce this overhead would result in much closer performance.

1. INTRODUCTION

1.1 Background

Since the dawn of history, humanity has been driven to improve his communications; striving always to exchange information faster and farther. As this century draws to a close, we are poised to take a significant step along this path with the advent of truly global personal communications.

The lure of ubiquitous global communications has lead to several large-scale commercial efforts in the US and abroad to loft multi-satellite networks in Low Earth Orbit (LEO) to provide mobile communications from any point on the earth. To reduce propagation delay, these systems attempt to minimize the number of links between the ground and space based portions of the network. Towards this end, several of the projects intend to use inter-satellite links to route the circuits completely in the orbital environment. The large velocity of the satellites with respect to the ground and one another, however, gives rise to high demands upon the circuit setup routine as it attempts to find the best path through a rapidly changing network topology.

1.2 The Problem

While the majority of the current proposals deal with voice traffic in a circuit switched metaphor, at least one project will target data traffic. The addition of packet switching capability compounds the difficulty of the space-based routing algorithm, as it attempts to dynamically optimize paths through the LEO network links. Little is known about the ability of current routing protocols, designed for a relatively static terrestrial network topology, to adapt to the constantly changing configuration of the space based links. Most especially, the ability of the routers to converge upon a satisfactory network state within this environment is critical.

This thesis attempts to shed light on the tradeoffs encountered for the problem of routing protocol selection. Because of the highly dynamic nature of these LEO constellations, selection of the routing algorithm can greatly affect the efficiency of the system. By studying the performance of two representative protocols under various operating conditions, we can begin to understand the advantages and drawbacks of each.

1.3 Scope

This study determines the speed of convergence, average packet traversal delay, and protocol overhead of two representative routing protocols (Extended Bellman-Ford and Darting). This is done on two different satellite configurations (roughly based on the current plans for the Iridium and Globalstar systems). The thesis makes no attempt to model or account for link quality, or other atmospheric and electromagnetic degradation effects.

1.4 Approach

Currently existing LEO network simulation models for the two satellite topologies mentioned above were modified to allow insertion of differing routing protocols. Additionally, the models, which approximated network convergence by using only one routing device, were extended to allow autonomous routing processes to exist on each satellite node. Each of the protocols was tested on the satellite networks at various traffic intensities using the BoNES Designer and SatLab simulation tools. Several runs were accomplished to increase the confidence in the results.

2. PRIOR WORK

Most of the commercial satellite communication (SATCOM) networks are only a few years away from operation, so it is no surprise that a good deal of research has been done on the structure and operation of Low Earth Orbit (LEO) SATCOM networks. This section provides a brief summary of the major challenges, and the various proposals for solutions that have been made. For organization, the chapter divides the topics up into sections corresponding to the OSI reference model.

2.1 Level 1 - Physical

For a global communications system to be a reality, the constellation must be visible from every point on the earth from which a customer might desire to make a connection. Presently, most companies place communication satellites in geosynchronous orbit. Unfortunately, at geosynchronous altitudes, propagation time of radio signals becomes a significant source of delay, and signal power requirements become large. This makes systems of "GEO" satellites somewhat less than desirable in global personal communications networks.

To combat the problems associated with geosynchronous altitudes, a satellite can be put into orbit much closer to the surface. While this solves the problems of signal power and delay, the satellite loses synchronization with a specific location, and can only see a much smaller area of the earth. To completely cover the surface, then, many more LEO satellites are necessary. However, because LEO satellites orbit below the Earth's Van-Allen radiation belts, they have the offsetting advantage of much cheaper construction.

Several studies have been done to determine constellations of LEO satellites that can ensure global coverage using an arbitrary number of satellites. Walker [Wal77] was among the first to propose such a system, and several commercial ventures have adopted the

Walker "delta" network. Walker constellations consist of several planes of inclined orbits with multiple satellites per plane. The system is described by six parameters that are chosen by the system designer to produce the desired degree of network coverage. For example, Globalstar (one of the leading LEO contenders) proposes to construct their constellation in a 48/8/1/52°/1389 format. This means that there will be 48 satellites in 8 different orbital planes, with a 7.5° phase shift between the planes. The orbital planes themselves will have an inclination of 52 degrees and the satellites will orbit at 1389 kilometers. For Globalstar, these numbers ensure that at latitudes below 55°, there is at least one satellite visible at an elevation of 40 degrees or greater [Rou93].

Another popular family of constellations is based upon the work of Adams and Rider [AdR87] who proposed the use of polar orbits to provide n-redundant global coverage. This is the orbital family used by Iridium, another leading network proposal. The constellation is based upon the idea of a "street of coverage" provided by each orbital plane. The designer can make any desired number of satellites visible from an arbitrary spot on the earth by simply "narrowing" the street (bringing the orbits closer together). Current plans for Iridium call for six 86.4° inclination orbits of 11 satellites each at an altitude of 780km [WuM94]. Minimum viewing elevation provided by this configuration would be about 10 degrees [Leo91].

One system (Teledesic by Teledesic Corp.) even proposes to use a sun-synchronous orbital configuration. Sun synchronous orbits are retrograde orbits. This means that their angle of inclination is greater than 90 degrees, so they seem to orbit backwards as viewed from the surface. If the inclination and other orbital parameters are chosen correctly, the satellite will maintain a constant position relative to the sun. This leads to benefits in constant time-of-day traffic scheduling [WuM94]. In other ways, this constellation type is very similar to the Adams and Rider family. Teledesic plans to use 21 orbital planes with a 98.2° inclination spaced about 9.5 degrees apart at an altitude of 700km. Each plane will

hold 40 satellites, providing access to two satellites at almost all times, one of which will always be available at a minimum 40° elevation [Tuc93].

2.2 Level 2 - Link Control

Once the satellites are in place, and the necessary frequencies obtained, the problem becomes how to organize information exchange over the communication links. Most of these issues have been tackled for terrestrial networks, but the unique characteristics of the orbital environment make some techniques more useful than others, and in some cases require completely new approaches.

Binder, et. al., propose a synchronous slotted approach in [BiH87], where the authors study the performance of inter-satellite crosslinks in a 240 vehicle constellation. First, they note that because the propagation times in orbit are generally much longer than packet transmission times, carrier sensing protocols are not useful. Instead, the authors develop a new approach that they call Pseudo-Random Scheduling (PRS). This method is applied on a pairwise basis for each crosslink, allowing freedom from the requirements of coordinating global synchronization.

PRS dynamically forms crosslinks as each satellite comes into range. In the method described, a satellite is assumed to be able to predict its own orbital position, but have no knowledge of the position of other satellites. Each satellite is also assumed to have an omni-directional listen and transmit capability. The omnidirectional mode is used to send special "hello" packets during idle times that contain the sender's ID, current position and motion, local clock time, and its random number seed. If a satellite listening in omni mode receives a hello packet, and decides based on the information received that it wants to establish a link, it responds to the hello packet with a similar reply. This packet is transmitted on a directional beam formed with the information in the original hello. The originator, on reception of the reply packet, then completes the handshaking by replying with a directional beam of its own.

Turning to the characteristics of the up and down links, transmitters in the proposed LEO SATCOM networks may occupy several different cells during any single call. Unlike terrestrial cellular networks, here it is the cells that are moving with respect to the (relatively) stationary transmitters. It is of interest then to determine how frequently the average user can expect to encounter handoffs and the effect these handoffs will have on the quality of the call and performance of the network.

The authors in [GaG94] derive an expression for the average number of handoffs experienced by a particular user based upon the speed of the LEO constellation and the radius of the satellite footprints. The derived relationship is:

$$\bar{h} = \frac{\left(3+2\sqrt{3}\right)\cdot V_l}{r\mu} \tag{2.1}$$

where V_l is the LEO speed with respect to the surface ($V_l = \frac{\omega \times R_g^{3/2}}{\sqrt{R_l}}$),

R_l is the LEO orbit radius, and R_g is the Geosynchronous orbit radius. From this it can be seen that the expected number of handoffs is a linear function of the footprint radius (r) and velocity of the constellation (V_l), times the average duration of a call ($1/\mu$).

2.3 Level 3 - Routing

If the satellites in our network were perfectly stationary, the questions of routing could be easily answered by any number of currently existing protocols. However, systems give up this ability when they chose to use LEO satellites to capitalize on their lower propagation and power characteristics. Thus, the question becomes, how do we properly route data packets in a network where there is no fixed relation between routing nodes and end-user devices? Each of the commercial efforts currently underway has plans for some type of proprietary routing protocol to address these issues, but published literature on these techniques is surprisingly sparse. A few teams have addressed specific issues though, and

some relevant information from the Strategic Defense Initiative (SDI) program has also been released.

The problem of determining the shortest path through the network given multiple constraints is not trivial. Jaffe [Jaf84] has proposed an algorithm that solves shortest-path multiple constraints in $O(n^5 b \log nb)$ time [$O(n^4 b \log nb)$ time per node in a distributed implementation] and $O(nb)$ space per node pair. Here n is the number of nodes and b is the largest value of the other constraints. He also presents three approximations to the solution that run in polynomial time and produce paths no worse than 2, 1.62, or 1.5 times the optimal solution.

Jaffe's method works by simultaneously calculating all the possible shortest distance paths between each node pair. Once the initial tables are calculated, subsequent routing decisions are simple table-lookups, thus amortizing the high initial runtime. A possible refinement whereby table entries that provide little value are dropped in order to economize memory requirements is also discussed. Additionally, Kung and Shacham present a somewhat simpler algorithm in [KuS84] that is useful in a centralized or semi-centralized environment and runs in $O(n^2 m T_1 \ldots T_m)$ time. (T_x is the value of the corresponding constraint.)

Several authors [BaA93], [ArA94] have looked at using the neural network "mean field annealing" technique to solve the routing problem for circuit switched networks. Circuit switched networks are typically constructed to allow alternative routes to be chosen for a transaction if the direct path is unavailable for some reason.[1] While this results in improved

---

[1] Almost every circuit switched routing proposal is based upon a modification to AT&T's Dynamic Non-Hierarchical Routing protocol used in their long-haul network. DNHR is applied to the top-level mesh-connected circuit switched network in such a way as to assure that at most two links are used to complete every call. When a call is placed, a specific prioritized set of paths is chosen based upon the date and time of day. Then, when determining the circuit setup, the direct path is checked for an available link first. If one does not exist, the various "alt-routes" are investigated to attempt to place the call. If these are also busy, the call is blocked. Alt-routes are chosen to involve only one other intermediary node, and the routing choice sets are optimized off-line and downloaded to the routers periodically based upon network usage patterns. [Ash90]

performance at moderate loads, it degrades quickly at higher traffic levels. The proposed solution involves reserving a fixed amount of bandwidth on each link for direct-routed calls only. The problem then becomes choosing this reservation value on a dynamic basis to best optimize the network.

As a solution, the technique of mean field annealing is applied to the network to generate routing maps that globally minimize the total call-block rate of the network, and maximize total throughput. A controller module monitors the network performance and determines when a new map needs to be calculated. Neural networks are composed of networks of simple linear operators that take an input, and based upon an "energy" function, provide an output. Through the use of feedback, the network converges to a solution. Energy functions are composed of a cost term, (which in this case is the total block rate of the network), and a constraint term that penalizes the cost if applicable constraints are violated. For example, constraints for the circuit switched case could specify that each node can only be assigned one reservation parameter, and that the total number of "on" neurons be equal to the number of network links.

Once the neural net is constructed, annealing works by repeatedly computing the network costs after perturbing the input slightly and keeping an updated entry for the current minimum values. In this manner, an entire network map indicating the shortest paths between each node pair may be constructed. To prevent the solution from being caught in a local minimum, non-optimal changes are allowed with a finite probability during each iteration. This effectively allows the simulation to "back-out" of a local minimum. Details can be found in [ArA94].

With the algorithms mentioned above, and others similar to them, it is possible to determine the optimal routing path for a given data packet. However, conventional flooding-type routing algorithms are not well suited to the orbital environment due to the large number of overhead messages they generate. This, coupled with long link propagation times, may result in transient loops forming in the network while a topology update is in

progress. The authors in [GrZ89] have investigated the performance of conventional Ford/Fulkerson and Merlin/Segall routing algorithms in SATCOM networks.

Ford/Fulkerson operates by having each node maintain a table of costs for all destinations reachable through its outgoing links. Upon detecting a change in one of these links, a node will send a control message to each of its neighbors. These neighbors will in turn update their own internal tables with the new information and pass the update farther along the network. The advantages of this method are its simplicity and asynchronicity. Disadvantages are slow convergence time and susceptibility to looping.

In contrast, the Marlin/Segall method is designed to prevent formation of these transient loops during a network update, but it does so at the expense of a slower convergence rate. In Merlin/Segall, all paths are stored as directed trees rooted at the sink node, which prevents loop formation due to the acyclic nature of trees. Updates begin at the sink node and propagate up-tree until the farthest node hears from all its neighbors. This farthest-distance estimate is then returned down-tree, with each node updating it's shortest-path entry upon reception of the return packet. Multiple update cycles may be required in the presence of network node failures.

The authors compared each algorithm on a hybrid LEO/GEO network of 18 and 6 nodes respectively. They found that the extra loop-preventing overhead introduced in Merlin/Segall led to performance an order of magnitude slower than the Ford/Fulkerson method.

The authors of [CaA87] present another type of loop-free algorithm that they designed specifically for the orbital environment. Created for SDI, its goals were distributed execution, robust recovery from massive failures, rapid adaptation to frequent load fluctuations and connectivity changes, and low delay with optimal throughput. The underlying topology for the network is assumed to be hierarchical, with a backbone of no more than 100 nodes serving clusters of second-level devices. The algorithm operates on

the backbone nodes and runs in $O(n^2)$ time with $O(n)$ messages per link. The algorithm uses a constrained flooding algorithm to broadcast "local" status periodically, with complete topology information being disseminated at longer intervals. This network information also includes congestion statistics. Load sharing among multiple paths to the destination is performed using a heuristic. Routing updates are synchronous (which is used to guarantee absence of transient loops), and occur approximately every 5 seconds. In the event of satellite failure, special "NO_PATH" messages are used to speed re-convergence of local tables in response to the failure.

A drawback of the preceding methods is the relatively high overhead associated with the control traffic. Tsai and Ma [TsM95] present a novel approach that they term "Darting" to overcome the high message overhead involved with flooding-type algorithms. The key idea behind Darting is to postpone transmission of topology update messages until it becomes necessary to actually transmit a data message. Darting uses two update mechanisms, which are triggered with the presence of a data packet. One mechanism updates the downstream nodes (i.e. "successor" nodes that the data packet will be visiting shortly in the future) and the other mechanism updates the upstream nodes ("predecessor" nodes.) The predecessor mechanism is triggered when the local node detects a discrepancy in topology views between itself and its immediate predecessor. When invoked, the mechanism send a special update packet backwards along the data path to pass updated network information to the sender. Successor updates are carried out by embedding all recent local topology changes in the outgoing message, which serves to propagate network changes downstream along the data path. By embedding local topology changes in each passing data packet, Darting eventually disseminates topology changes throughout the network.

Unlike conventional flooding algorithms, which exchange periodic control messages to prevent the formation of "message traps" (i.e. routing loops), Darting concentrates on dynamically breaking any traps that have formed. This eliminates the need to exchange update messages on a regular basis. When a source node desires to transmit a packet to a neighbor, it consults its routing tables, and places the anticipated cost of delivering the

message to the destination in a header field. It then calculates the anticipated cost for its neighbor to transmit the message to the destination, and places that in another header field. Upon receiving the packet, the neighbor node can use this data to determine if the sender is using current routing information. If a discrepancy is detected, the predecessor update mechanism is invoked. Normally this mechanism only updates the immediate predecessor. Optionally, though, predecessor update messages can be allowed to propagate farther back upstream before being discarded. This will result in faster network convergence at the expense of additional control traffic overhead.

Shacham [Sha88] conducts a detailed look at the obstacles and hurdles that the network and transport level protocols will encounter in orbit. He proposes using the predictability of short-term changes in the local environment to reduce the amount of topology-update traffic generated by the network. This is based upon the assumption that any single satellite will have information about the positions and velocities of its neighbors and be able to detect when two of them will come into communications range. It can then notify the network of the expected changes in the local environment for a short time into the future, and can do this at a much lower frequency than a simply reactive approach.

Shacham also looks at the problem of address binding. Because the satellites above any particular user are constantly changing, there is no fixed relationship between a specific user and network node. Thus, addresses based on geographical location of the end terminal are suggested, with the possibility of multi-homing, so that a packet can be routed to "any satellite above Washington DC", for example. Information on the locations of specific destinations would be stored in a distributed database indexed by destination ID. It is also suggested that satellites dynamically reduce topology complexity by only forming network links with a subset of available neighbors.

Finally, because of the combination of high link data rate with high propagation delay, the author recommends avoidance of the go-back-n type transport protocols in favor of selective repeat protocols such as VMTP. This avoids the unnecessary retransmission of

packets that may have been received correctly, which can be large due to the large window size required to handle the rate/delay combination, as discussed below.

2.4 Level 4 - Transport

While each of the link control methods of section two has some provision for error detection and retransmission, normal methods of automatic repeat requesting (ARQ) are not well suited to orbital links. This is primarily due the simultaneous occurrence of high data rates and high propagation times found in this environment. For conventional go-back-n ARQ protocols, the large window sizes required for this rate/propagation ratio lead to many needless packet retransmissions. Because of this problem, reliable transport provided by Level 4 protocols becomes increasingly important.

VMTP is a transport level protocol that has been suggested as a possible candidate for end-to-end reliable delivery over LEO SATCOM systems [Sha88]. It provides higher level processes with the facilities to conduct "conversations" between end nodes using special constructs termed "message transactions". Each high level conversation is built out of VMTP message transaction primitives. A message transaction is a request-response pair with reliable delivery on both the request and response messages. By replacing the conventional virtual circuit paradigm with these message transactions, VMTP is able to cut down the number of packets exchanged for simple operations like file query, get time, and basic remote procedure calls to a single request-response pair. This is in contrast to a virtual circuit based protocol like TCP that normally requires six or more packets for the same operations. VMTP also has provisions for packet duplication suppression even when the delay between the original and duplicate is relatively long.

Additionally, VMTP includes a method of dynamic transmission rate "throttling" via its selective repeat retransmission scheme. Each VMTP message is divided into a packet group with a bitmask field for the group placed in each packet header. Each bit in the bitmask corresponds to a packet in the group. Therefore if the receiver gets any portion of the packet group, it can easily indicate to the transmitter which packets to resend by simply

returning the mask with the corresponding bits set. In the case where the transmission rate is too high for the network, upon analyzing the returned bitmask, the sender may discover that every k^{th} packet is being lost. It can then increase the amount of time between each packet in the group to accommodate current network conditions and reduce subsequent re-transmissions due to buffer overruns [Che86]. It is highly preferable for the transmitter to pace itself rather than waste network bandwidth and satellite power in needless retransmissions.

2.5 Current System Proposals

Three proposed LEO systems seem to be emerging as the top contenders to actually field a workable communications system. These are *Iridium* by Motorola, *Globalstar* by Loral/QUALCOMM, and *Teledesic* by Teledesic Corp. (Formerly known as the Calling network.) Each has chosen slightly different implementations (the structure of each constellation was detailed in section 2.1.)

Globalstar and Iridium plan to focus primarily on voice and facsimile traffic in the 4800 baud range, while Teledesic intends to provide multiple 16kbps data channels. The former are therefore primarily circuit switched systems while the latter is more similar to conventional computer networks. Indeed, as mentioned before, Globalstar does not even intend on employing satellite cross-links, instead routing all traffic to regional ground-stations and then relying on conventional land-lines to complete the circuit. This, in turn, has probably motivated Globalstar's higher altitude (1400km vs. 700km), which results in a larger satellite footprint, reducing the number of ground-stations required [Wis95].

The crosslink structures of Iridium and Teledesic, and the pattern of cells they lay down on the surface, reflect the differences in emphasis in the two systems. The four crosslinks in Iridium are partitioned into 1300 fixed channels, while Teledesic uses 8 crosslinks of 138Mbps each. While both constellations orbit at approximately the same altitude, the much higher data rates envisioned by Teledesic require it to have an order of magnitude more satellites than Iridium. Correspondingly, each satellite footprint in Teledesic is much

smaller: 1400km vs. 4700km. Additionally, Teledesic will employ a much higher elevation angle, 40 degrees vs. 10, to improve signal performance [Leo91], [Tuc93].

2.6 Performance Studies

None of the systems mentioned above will be operational until at least 1998. Until then, the only method we have for examining performance of these types of systems is computer simulation. The authors of [TsC94] studied the performance of a hybrid LEO/GEO network for circuit switched traffic. The LEO network was similar in dimension and construction to Iridium, with the addition of three Geosynchronous satellites that were used as alt-routes for overflow traffic. Each LEO satellite had 6 cross links to neighboring LEO satellites, plus one link to the nearest GEO satellite. Routing was performed with traditional DNHR style algorithms, with calls alt-routed through the GEO satellites if the proposed LEO route exceeds a threshold hop-count, or if a LEO link in the path is saturated. Residual capacity reservations were used on the crosslinks to improve stability. Results indicated a blocking probability of less than .6 at a load of 1000 Erlangs. A measurable improvement was found to exist from inclusion of GEO satellites to the network. The improvement is highly dependent on the value chosen for the hopcount threshold though, and would need to be dynamically tuned to assure optimal network performance. For the Iridium-type system simulated, the ideal threshold was approximately 4 hops.

2.7 Summary

Current efforts at fielding global personal communication networks seem well underway, with no significant technical obstacles remaining. However, while information regarding the link access methods of each system has been relatively well documented, not much has been officially published regarding the routing methods these systems intend to employ. The problems of routing in an orbital environment are demonstrably more complex than an earth-based system. While several authors have proposed tools to handle various parts of the issue, very little comparison of the relative merits of each solution has been accomplished. Until the commercial systems are fielded, we must fall back on simulated results for the initial answers.

3. METHODOLOGY

3.1 Introduction

As discussed in the previous chapter, very little information is available on the relative merits of the various approaches to the orbital routing problem. To attempt to alleviate this problem, this thesis provides a comparison of the performance of two different routing protocols under various orbital and operational characteristics. This was done using commercially available simulation tools as described in the remainder of this chapter.

3.2 Experiment Design

3.2.1 Choice of Method. There are three primary methods of obtaining measures of network performance. You can build the system in question and measure it in operation. You can construct an analytic model and study the mathematical representation of the system. Or you can build a simulation model and run experiments.

Simulation has been chosen for the following reasons: 1.) Though analytic modeling provides a more accurate description of a network (if you can find a solution), the size of the networks in this thesis makes analytic solution intractable. Kleinrock observes "When one relaxes the Markovian assumption on arrivals and/or service times, then extreme complexity in the interdeparture process arises ..." Even if we were to make the simplifying assumption of Poisson arrivals and exponential service times, analytic solution for equilibrium of a closed network of queues involves solving $\binom{N+K-1}{N-1}$ simultaneous equations[2]. Even for the smallest proposed constellation (Globalstar with 48 nodes), solving the system for only 5 packets in the network requires solution of almost 2.6 x 10^6 equations [Kle75]. 2.) Currently none of the proposed satellite systems are operational, making study of a physical system impossible.

[2] N is the number of nodes and K is the number of packets in the network.

3.2.2 *Choice of Protocols.* Original plans called for a comparative performance analysis of multiple orbital protocols. However, at the present time, there is surprisingly little published about routing protocols to be used in these LEO systems. Darting was the only published protocol asserted to be suitable for operation in satellite systems. Each of the commercial systems discussed in the previous chapter plans on employing some sort of proprietary protocol, and says little else. Accordingly, it was decided to use a representative terrestrial protocol to gain a baseline against which Darting could be compared. The terrestrial protocol selected is an extension to the venerable Bellman-Ford protocol proposed by Cheng, et. al. [ChR89].

This Extended Bellman-Ford algorithm (exBF for short), fixes the counting-to-infinity behavior and bouncing effect of the original Bellman-ford protocol. Originally used in the early days of the Internet as the inter-gateway protocol, it was these two problems which led to Bellman-Ford's replacement with more modern protocols. A comparison at the University of Maryland [ShA92] showed that with Cheng's extensions, Extended Bellman-Ford performs comparably with the newer systems. Therefore, due to relative simplicity and universal familiarity of Bellman-Ford, this protocol was selected for the comparison.

Cheng solved the bouncing and counting-to-infinity problems by noting that those effects were caused by routers advertising what he terms non-"simple" paths. By this, he means one router telling a neighbor that it has a path to a destination when the path so advertised goes through that neighbor. By adding an extra field to each entry in the routing table, Cheng is able to detect these non-simple paths and modify router advertisements accordingly. It should be noted that only the basic version of Cheng's extensions are implemented for this thesis. The synchronization protocol enhancement also proposed by Cheng which would eliminate short-term looping effects was not implemented.

3.2.3 *Simulation Construction.* The network simulations in this thesis were built using two packages, BoNES Designer and SatLab, published by Cadence software. Designer is a top-down block-oriented network simulation package and SatLab is a satellite constellation simulation and optimization package. SatLab is used to communicate the relative positioning and visibility information of each network node to Designer, and comes with the Globalstar and Iridium constellations built in. The example satellite communication system provided with Designer was modified and extended to fit the needs of this thesis as described below.

Following the top-down approach, a copy of the top level schematic was made for each routing protocol and modified as shown in Figure 1 and Figure 2. Text labeled with a "P" is a simulation parameter set at runtime. Text labeled with an "M" is a memory variable, and the text labeled with an "R" is a FIFO queuing structure. The values of the link-rate parameters are taken from the proposal for the Iridium system [FCC91] and are 12.5

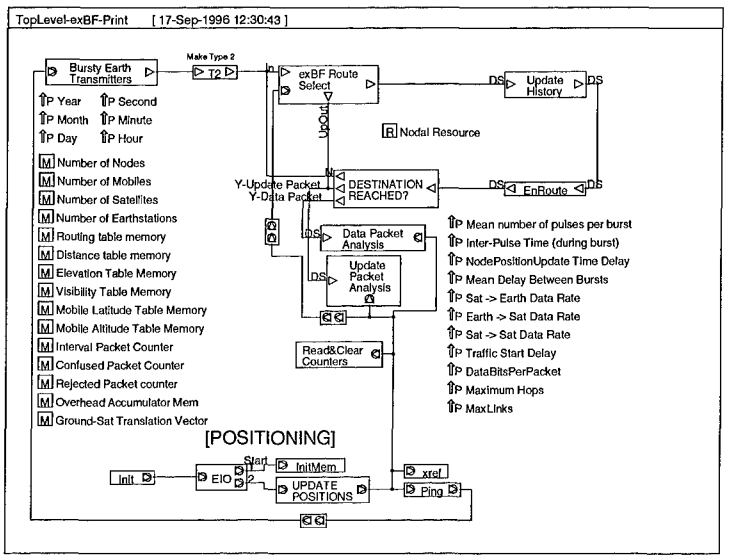

Figure 1: Top level Extended Bellman-Ford schematic

megabits per second for the earth to space links, and 25 megabits per second for the inter-satellite links. Complete details on the parameter values and memory variable functions can be found in Appendix A.

Simulation begins with the Init block in the lower left corner. It initializes some memory variables and obtains the satellite positions from SatLab. Once this is done, the traffic generators are pinged to begin transmitting. Simultaneously, the ground-station to satellite cross-reference table is created, and the routing and analysis blocks are initialized.

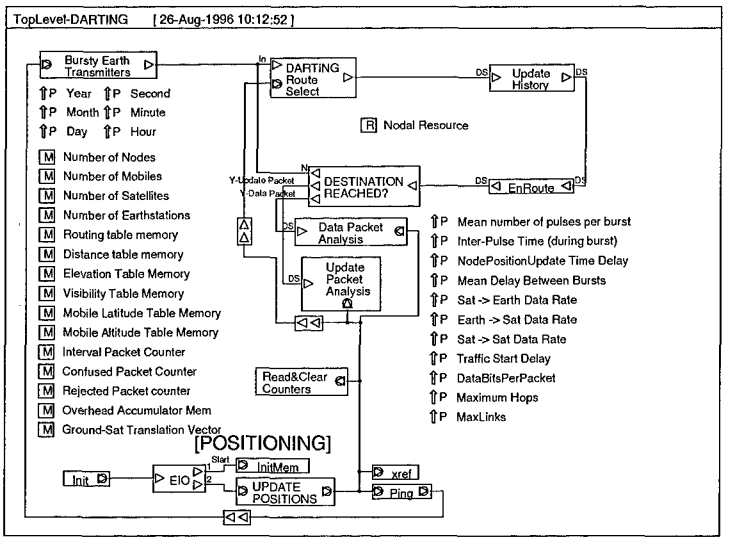

Figure 2: Top level Darting schematic

Data packets flow from the transmitters into a routing node, which is contained in a satellite. The routing block looks at the destination of the data packet and determines the appropriate next hop, updating the packet fields accordingly. The packet then passes into the Update History block which maintains a list of every node the packet has traversed. It then passes to the EnRoute block, which examines the current and next nodes and delays

the packet for a calculated amount of time to simulate queuing, transmission, and propagation delays.

The simulation then checks to see if the packet has reached its destination. If it has, it then passes to the analysis blocks, which calculate the performance statistics for the routing protocol. If the packet has not reached its destination, it passes back to the routing routine to begin another circuit. Complete details of the simulation sub-blocks can also be found in Appendix A.

Two routing protocols, Extended Bellman-Ford and Darting were chosen for simulation. Bellman-Ford was chosen due to its use in the early Internet, and because it is representative of the Distance Vector class of routing algorithms. The version of Bellman-Ford used here is an extension to the original protocol proposed by Cheng, et. al., and eliminates the counting-to-infinity problem of the original [ChR89]. Darting was selected as the second protocol because the authors assert its suitability for the LEO environment. However, during verification of the simulation model, a weakness in the protocol was encountered when attempting to handle the non-uniform traffic distribution of the network. Some modifications were therefore made to the protocol to overcome this problem, and are detailed in Appendix C.

These protocols were simulated on the Iridium and Globalstar constellations because they are representative of the two main constellation families and were provided with the SatLab simulation package as shown in Figure 4 and Figure 3. These figures also show the initial state of the inter-satellite links chosen by the routing protocols. The Iridium constellation has a periodicity of under 15 minutes. It was therefore decided to run each simulation for a total of 960 seconds, with the first 60 seconds being discarded to allow initial transients to die out. Satellite positional updates were arbitrarily set at one minute intervals.

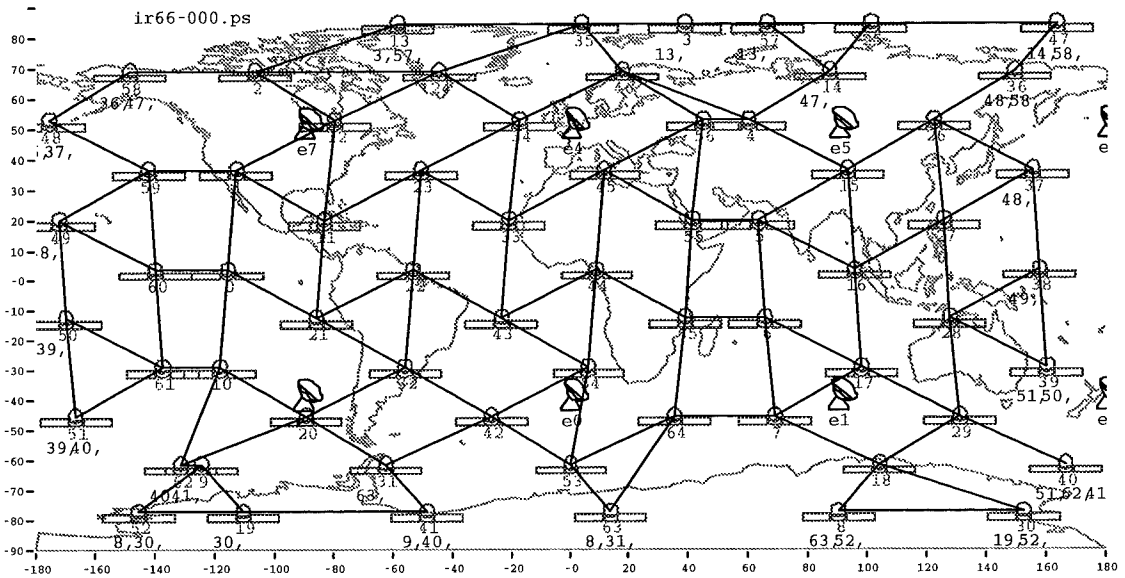

Figure 3: Iridium Constellation

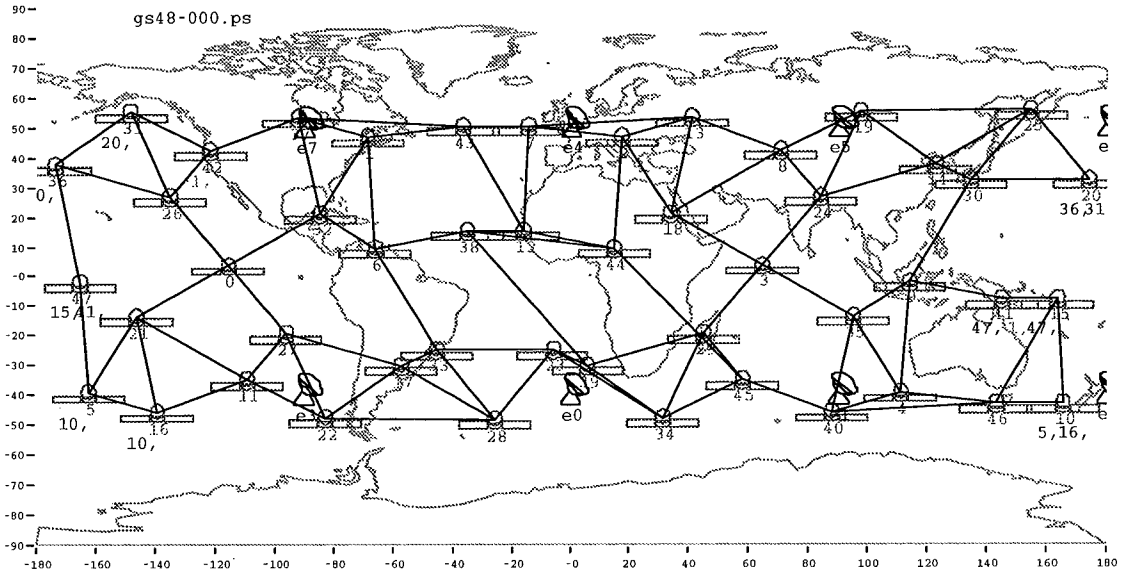

Figure 4: Globalstar Constellation

3.3 Data Gathering

The parameters of interest for this thesis are packet traversal time, protocol convergence rate, and protocol overhead. The simulation calculates these parameters for each satellite update period, and several independent simulation runs are made for each configuration to improve the confidence level in the results.

Packet traversal time was selected because the main benefit of LEO networks is the ability to conduct "real time" (less than 400ms) transactions. Thus the impact of each protocol on traversal time is of great interest.

Convergence rate was studied because it is a dominant factor in determining how efficiently a protocol moves data through the network. Protocols that determine the optimal paths faster have a significant edge in providing lower average traversal times in the dynamic LEO environment.

As was mentioned in Chapter 2, conventional flooding algorithms achieve rapid convergence rates through a high protocol overhead. Therefore the amount of overhead load inflicted on the network by each protocol is of interest in determining how much of a penalty is imposed by each algorithm in obtaining its convergence rates and traversal times.

The simulation measures packet traversal time by accumulating all calculated delays encountered by a packet in one of the packet's fields. Each time the packet encounters a queuing, transmission, processing, or propagation delay, the duration of the delay is added to the delay field. When the packet reaches its destination, this value is recorded for later analysis.

Similarly, overhead is measured by maintaining a packet length field, and accumulating any additional overhead (if any) added by each node the packet passes through. When the packet reaches its destination, the analysis blocks record the number of data bits and the number of overhead bits contained in each packet. Packets that contain only routing update information are counted as completely overhead.

Convergence rate is determined by monitoring the network for the presence of routing update packets. A sub-section of the update packet analysis block measures the time elapsed between delivery of update packets to their destinations. A memory variable local to the analysis section is used to accumulate these times. The convergence time for each protocol is then taken to be the total elapsed time from the last satellite position update to the most recent delivery of an update packet. In other words, each protocol is considered to have converged when no more update packets are present in the network.[3]

3.4 Operational Assumptions

Several simplifying assumptions were made to allow a closer focus on the parameters of interest in the project:

1. Communication links are error-free: Because we are primarily interested in the routing (OSI level 3) performance of the systems, it was decided not to model error handling and recovery on the links, as this task is primarily handled at levels 2 and 4. From a level 3 perspective, addition of a finite error probability on each link would only have the effect of increasing the rate of traffic arrival. Since each of the systems was modeled with various source traffic intensities, addition of error recovery traffic was deemed unnecessary.

2. Uniform source distribution: Complete uniformity for source distribution is assumed. The commercial systems proposed address two purposes for operation. They propose to provide global mobile communications, plus primary communication for underdeveloped areas of the world (fill-in service.) We would expect most of the source traffic for the mobile users to begin or end in one of the larger metropolitan areas, but that the traffic for the fill-in service would have a more uniform distribution. Because it is uncertain at this time which function will provide the majority of the traffic, and what percentage of the mobile traffic will come from which areas of the globe, a uniform distribution is used here.

---

[3] Note that while the absence of update packets from the network in itself does not guarantee that the protocol has converged to an *optimal* solution; it is not the purpose of this thesis to evaluate the optimality of each protocol, but to compare their relative performances. Any non-optimal solution will increase average traversal time, and thus penalize the protocol in the final analysis.

Due to memory limitations in the Sun workstations, these sources were limited to one per globe octant (8 total).

3. Uniform destination distribution: While it is known [Cha89] that the probability of a particular destination site decreases as the distance from the source increases, modeling destination addresses with an exponential, rather than a uniform, distribution leads to localized islands of activity in the network. Because it was desired to exercise as many of the routing nodes as possible, a uniform distribution for destination addresses was also adopted.

4. Address binding is handled on a geographic basis as proposed by Shacham [Sha88]. Groundstation traffic is handled by the nearest network satellite. Actual address lookup at call setup is assumed to be handled by higher level protocols and is therefore ignored in the simulation.

5. Infinite buffers: Similarly to the assumption on error freedom, the addition of finite buffers to the simulation would only lead to an increase in data traffic as sources re-send rejected packets. Therefore, because the simulations were run at multiple loading levels anyway, finite buffers would not have added anything to the comparison.

During operation of the simulation, topology update packets are given queue priority over data packets. This choice increases the protocol convergence rate, which has the benefit of reducing extra sojourn time due to outdated routing information.

Simulation of orbital mechanics is handled by the SatLab simulation tool. SatLab handles orbital perturbations caused by the Earth's oblateness and other factors, as well as relative node positioning and field of view. SatLab assumes two nodes are obscured if they are separated by the surface or over 90 km of atmosphere [Sat95].

3.5 Verification / Validation

3.5.1 *Routing Algorithm Verification* Each routing algorithm was constructed in a specialized BDE[4] framework that mimicked the full simulation from the router's point of view. Each protocol was verified against three small test constellations for which the optimal spanning tree had been calculated by hand. After each satellite update period, the state of the network was dumped and compared with the optimal solution.

Once each protocol had successfully passed the test constellations, it was placed into the full simulation and run at a very low traffic intensity to verify that the protocol could operate with the full compliment of network and would converge to a solution.

3.5.2 *Designer BDE Verification* Initial modifications to the Designer SATCOM example system were carried out by Capt Doug Stenger [Ste96] for a parallel thesis effort; and first level verification of the BDE was accomplished there. Further modifications were enacted to accommodate the different source and destination setup in this thesis.

These further modifications were verified by single-stepping the simulation for each packet type through all possible simulation sub-paths. At each step, the actual response of the packet was verified against expectations and each packet delay verified against hand-calculated values.

3.5.3 *System Validation* Because there are currently no LEO systems in operation, validation consisted of attempting to keep as close as possible to the published system proposals. When a choice was available, the corresponding data value from the Iridium proposal [FCC91] was used. For instance, the satellite data rates, available crosslinks, and multiple data access delay were all taken from that document. Other values, such as the data packet size, were arbitrarily chosen based upon experience with reasonable network values.

---

[4] BDE stands for Block Diagram Editor, but here it is used to generically refer to the Designer simulation schematics that are edited using the Block Diagram Editor.

3.5.4 Summary Through the methods described above, a simulation of the LEO operating environment was constructed for each routing protocol. Multiple simulations runs at various loading levels and in different constellation families were executed to assess protocol performance. The results of those simulations can be found in Chapter 4.

4. RESULTS AND ANALYSIS

4.1 Accuracy

Any conclusions drawn from simulated data are only as good as that data. It is therefore important to be reasonably certain that the statistics drawn from the data are representative of the population. For this purpose, the method of determining confidence intervals with a specified precision as discussed by Banks, et. al. [BaC96] was employed.

Rearranging Bank's equation to solve for the confidence interval half-length given a fixed number of iterations, we obtain the following:

$$t_{\alpha/2} \leq \frac{\varepsilon\sqrt{R}}{S} \tag{4.1}$$

Here ε is the precision to which it is wished to obtain the confidence for the statistic, R is the number of independent replications, and S^2 is the sample variance. From equation 4.1, the maximum confidence of the data can be determined by integrating the density function for the Student-t distribution with the appropriate degrees of freedom and calculating the percentage area. The Student-t density function is:

$$f(t) = \frac{\Gamma\left(\frac{v+1}{2}\right)}{\Gamma\left(\frac{v}{2}\right)\sqrt{v\pi}}\left(1+\frac{t^2}{v}\right)^{-(v+1)/2} \tag{4.2}$$

Using numerical integration[5], table 1 lists the confidence levels obtained from the data for 1% accuracy on the mean. That is, the table indicates the relative certainty that the sample mean is within 1% of the true population mean. The number in parenthesis is the number of independent repetitions completed for that data. The Darting protocol shows such high convergence variance that even with extra repetitions confidence is low. Also, due to long simulation execution times, fewer iterations were completed at higher loads, and

[5] Via Matlab's Quad8 function.

the averaging of partial simulation runs of different lengths leads to somewhat artificially low confidence values.

Table 1: Data Confidence

Constellation	Protocol	Statistic	1% Load	10% Load	20% Load
Globalstar	Bellman-Ford	Convergence	0.92 (3)	0.97 (3)	0.14 (2.5)
		Delay	0.99 (3)	0.99 (3)	0.15 (2.5)
		Overhead	0.96 (3)	0.99 (3)	0.49 (2.5)
	Darting	Convergence	0.17 (11)	0.16 (3)	0.04 (1.5)
		Delay	1.00 (11)	0.83 (3)	0.19 (1.5)
		Overhead	0.99 (11)	0.99 (3)	0.18 (1.5)
Iridium	Bellman-Ford	Convergence	0.96 (3)	0.56 (1.5)	0.46 (1.5)
		Delay	0.98 (3)	0.98 (1.5)	0.81 (1.5)
		Overhead	0.97 (3)	0.87 (1.5)	0.91 (1.5)
	Darting	Convergence	0.09 (11)	0.01 (1.5)	0.01 (1.5)
		Delay	1.00 (11)	0.78 (1.5)	0.44 (1.5)
		Overhead	0.94 (11)	0.14 (1.5)	0.01 (1.5)

(Load percentages are defined with respect to the Iridium ground to space maximum data rate of 12.5 Mbps [FCC91].)

4.2 Simulation Mechanics

As mentioned in the previous section, difficulty in simulation execution was encountered at the higher loading levels. Indeed, it proved impossible to execute simulations within the 64 megabyte RAM constraints of the shared Sparc-20 workstations above a 20 percent load. Post simulation analysis (i.e. hindsight) showed this to be caused primarily by a sub-optimal packet design. (See Appendix A for details on the structure used.) The design was adapted from the demonstration satellite communication system provided with the Designer simulation package. It employed a shell-within-a-shell embedding structure the turned out to greatly increase the overhead associated with packet book-keeping. Measurement using a memory monitor showed about 16K per packet being used. With about 20 million packets being generated per simulation run, and roughly 210 accesses per packet along a typical route through the network, over 66 terabytes of information is being processed per run. With 80ns RAM in the Sparc-20s, this yields a run-time surprisingly close to the 7 week runtimes actually encountered. Additionally, output from Designer's profiling utility shows

over 50% of CPU time involved with inserting, removing, and type-converting packet information. While this excessive runtime led to some ragged run-lengths at the higher loads, the overall trends for the data are clear and are detailed below.

4.3 Individual Protocol Data

4.3.1 *Extended Bellman-Ford (Globalstar):* Figure 5 and Figure 6 below show the packet end-to-end delay characteristics of the Bellman-Ford Protocol over time. Discounting the

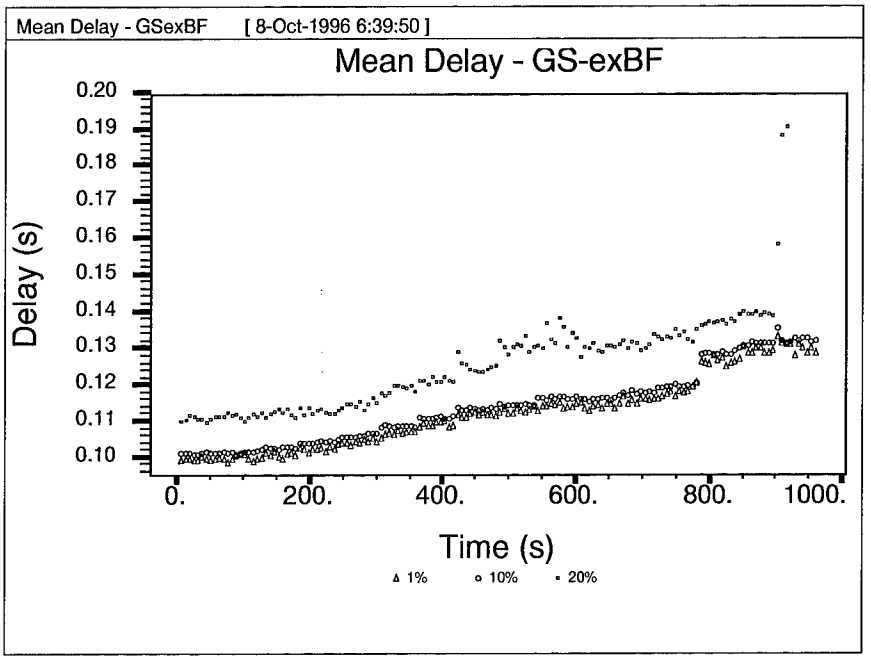

Figure 5: Bellman-Ford Mean Delay in Globalstar

first 60 seconds, Bellman-Ford shows an average delay time of 0.1128 seconds at a 1% load, 0.1147 seconds at 10%, and 0.1379 seconds at 20%. This is a 1.73% increase from 1% to 10%, and an 18.98% increase from 10% to 20%[6]. Maximum packet traversal delay averages 0.1687, 0.1842, and 0.3157 seconds respectively, or a 9.07% increase from 1% to

[6] Percentage changes are calculated by computing a delta for each pair of datapoints and calculating the percentage of the base value that delta represents. The number quoted in the text is the mean of these percentages, and thus may deviate slightly from the same calculation applied to the average numbers from the previous sentence. It is important to notice that because the percentages reference different bases, the 1-10% and 10-20% figures are *not directly comparable.*

10% and 69.56% from 10% to 20%. An interesting (and unexplained) feature of the graphs is the instability in the 20% runs during the 900 second update period. While the 1% and 10% numbers also show a spike at this time, they quickly return to steady state. This is not the case with the 20% runs, however. The rise in packet traversal time is accompanied by a large growth in the queue size of satellite #21. However, because the simulations require over 7 weeks to reach this point, extensive experimentation to discover the cause was infeasible. The most likely culprit is the loss of routing information packets in a transient loop that exceeded the maximum hop-count, preventing network convergence, as the records show several packets removed from the network at this point for that reason.

The increasing trend shown by the data is predominately caused by the increasing distance of the satellites from each other with time.[7] The slope of the increase reduces

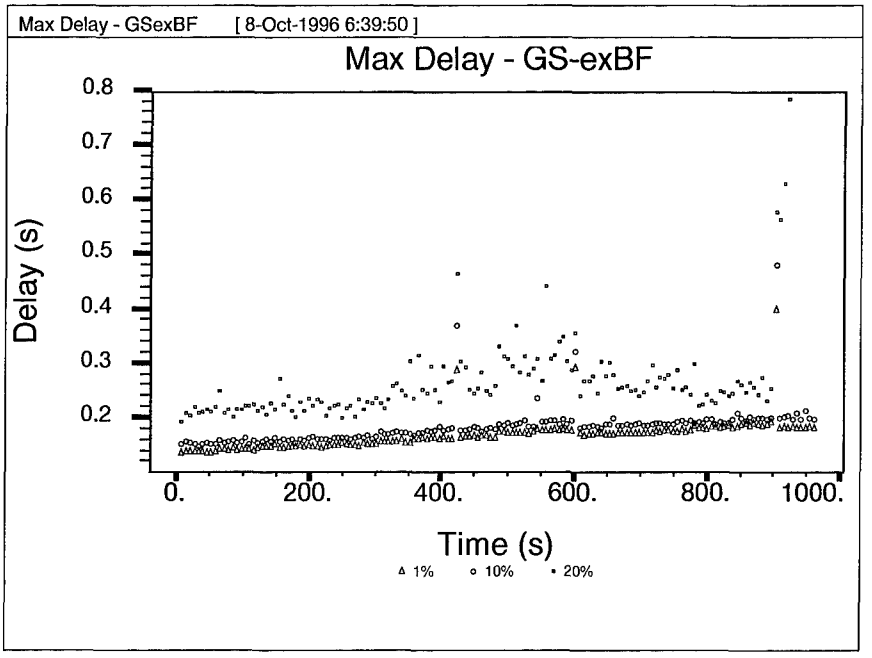

Figure 6: Bellman-Ford Maximum Delay in Globalstar

[7] Examining one of the paths in the Iridium constellation shows an average delta of 152km per 60 second update period between the groundstations and the uplink satellite, 80km between satellites in co-rotating planes, and 500km in counter-rotating planes. Disregarding everything except the up and down-links, that's 2 links * 152km * 1sec/3e8m = approx. 1 msec per update. This is of the same magnitude as the slopes seen in the graphs.

around 600 seconds because one of the routes has reconfigured to a shorter link at that point. Conversely, at around 800 seconds, a jump in the mean hop count indicates that one of the links along the preferred route has become inactive and required use of a longer path. Looking closely at the 10% mean delay data, it can be seen that delay within a satellite update period is fairly constant, with very noticeable stepping between updates in the 300-500 second range.

As the amount of traffic in the system increases, queuing delays begin to play a factor in delay, as can be seen in the higher mean delay and variability in the 20% data. Individual spikes corresponding to network re-configuration can be seen in the mean data at 422, 486, and 546 seconds. Similar performance is seen in the maximum delay data.

Convergence time for Bellman-Ford in Globalstar is fairly constant across the entire simulation, with peaks of activity at 420, 600, and 660 seconds due to several network links re-configuring at those times (Figure 7). The first two data points are discarded as startup

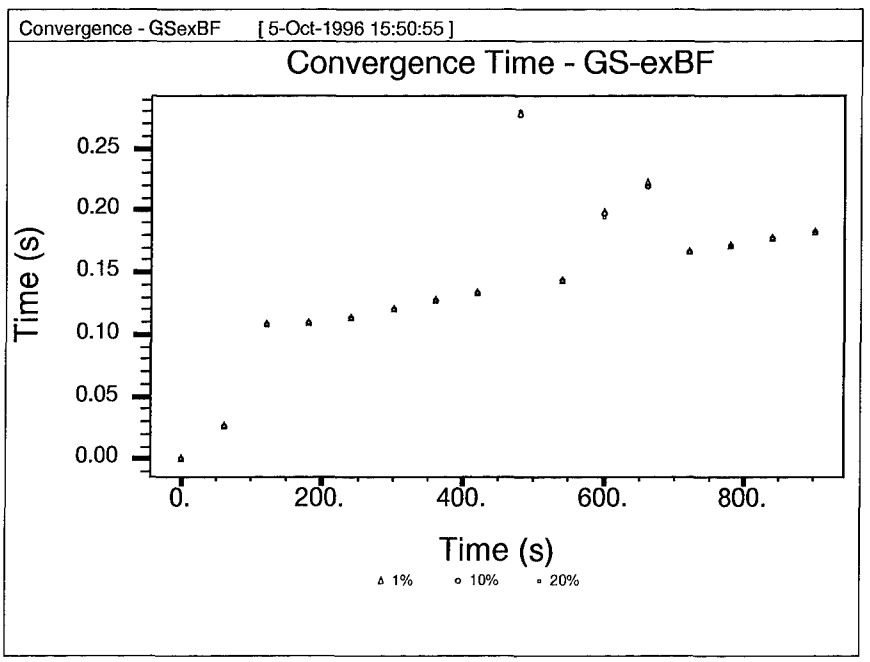

Figure 7: Bellman-Ford Convergence Time in Globalstar

transients (the protocol takes several shortcuts during that period to diminish required simulation time). The slope of the trend line is due to the increasing propagation time mentioned above.

Apart from the general trends, convergence time averages 0.1609 seconds at 1% load, 0.1606 seconds at 10%, and 0.1605 seconds at 20%. This corresponds to a 0.19% change from 1 to 10% and a 0.26% change from 10 to 20%.

Regarding overhead performance (Figure 8), the data shows no real surprises. Keeping in mind the fixed packet header overhead required to transmit data (30.72, 153.6, and 307.2 Mbits), the overhead required to converge the constellation is constant with respect to loading level and increases significantly only during times of increased convergence activity. Specifically, overhead traffic accounts for 48.05% of total traffic at 1% load, 24.29% at 10%, and 22.21% at 20%.

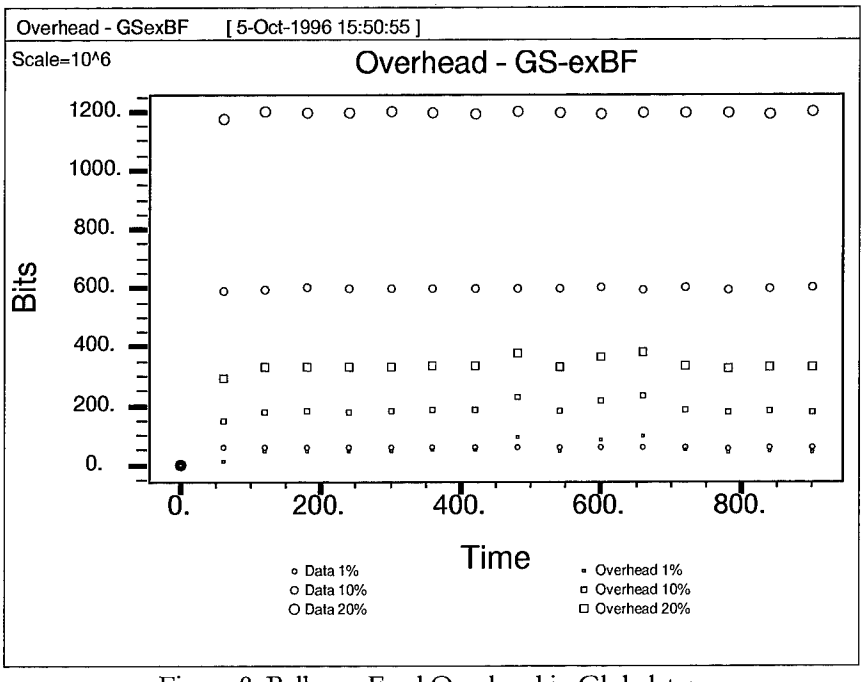

Figure 8: Bellman-Ford Overhead in Globalstar

4.3.2 Bellman-Ford (Iridium): Bellman-Ford's performance under Iridium is roughly equivalent to Globalstar, obtaining slightly faster traversal time at the expense of marginally higher overhead. This is most likely due to the shorter average link length under Iridium due to its larger number of satellites. Under low loads, there is actually more overhead traffic than data traffic traversing the network.

As shown in Figure 9 and Figure 10, data traversal times average 0.1003 seconds at 1%, 0.0970 seconds at 10%, and 0.1078 seconds at 20% (2.17% and 11.18% increases respectively) and maximum observed delays averaged 0.2129 seconds at 1%, 0.1709 seconds at 10%, and 0.2441 seconds at 20% (12.62% and 47.46% increases.) Spikes are visible at the more active update times, with a transient routing table loop causing a large spike at 420 seconds.[8]

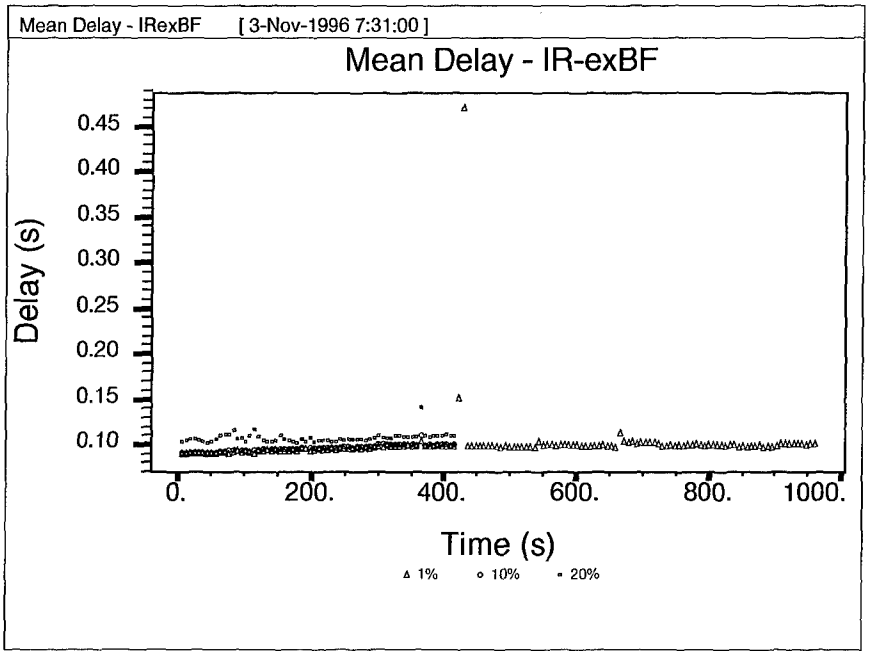

Figure 9: Bellman-Ford Mean Delay in Iridium

[8] Cheng has also proposed a more complex version of Extended Bellman-Ford which eliminates these transient loops.

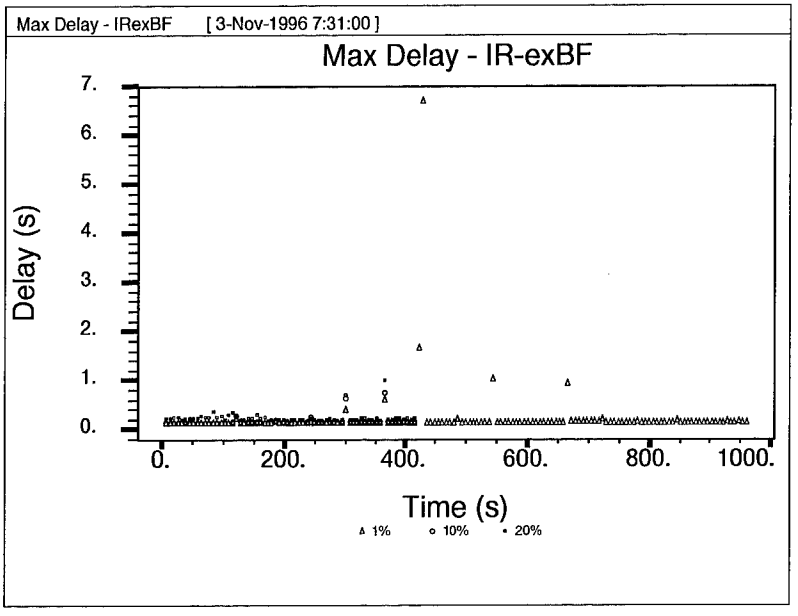

Figure 10: Bellman-Ford Max Delay in Iridium

Convergence times for Iridium (Figure 11) show more variability than Globalstar due to higher reconfiguration activity in Iridium (links begin to reconfigure at 180 seconds as opposed to 420 seconds in Globalstar). The large increase at 420 seconds is due to the packet spike mentioned above. Convergence times average 0.9894 seconds at 1%, 0.4389

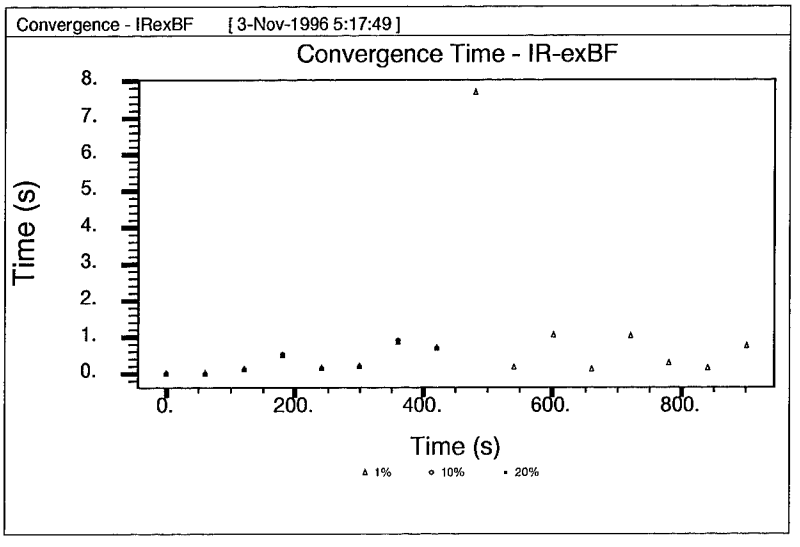

Figure 11: Bellman-Ford Convergence in Iridium

seconds at 10% and 0.4233 seconds at 20% (3.26% and 5.20% increases.)

Overhead accounts for 77.05% of total traffic at 1% load, 35.79% at 10%, and 28.84% at 20% load (see Figure 12).

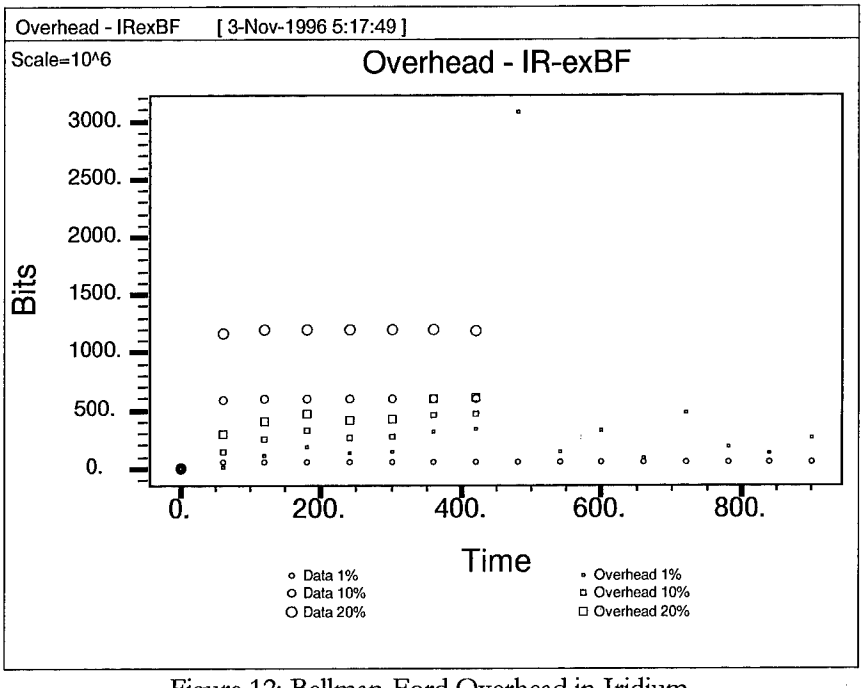

Figure 12: Bellman-Ford Overhead in Iridium

4.3.3 *Darting (Globalstar):* The delay results for Darting in Globalstar show visible spikes at almost every positional update (Figure 13 and Figure 14). These are most likely caused by surges of routing update packets as the network attempts to re-converge. Average delay is 0.1132 seconds at 1% load, 0.1174 seconds at 10%, and 0.1319 seconds at 20% (3.68% and 17.48% increases). Maximum packet delays averaged 0.2442, 0.2968, and 0.4280 seconds respectively (25.59% and 57.19% increases.)

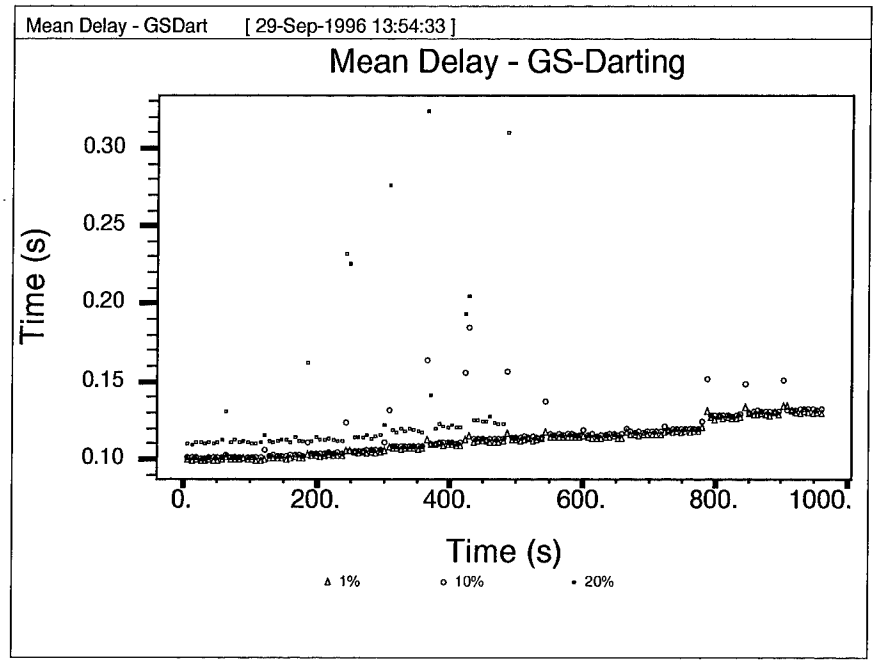

Figure 13: Darting Mean Delay in Globalstar

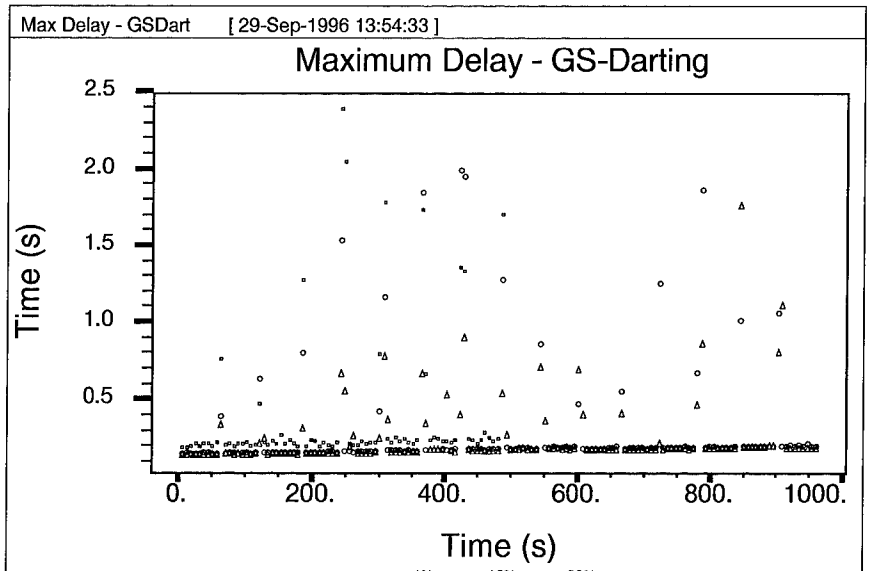

Figure 14: Darting Maximum Delay in Globalstar

In Figure 15, convergence time shows an inverse relationship to data rate. At the 1% loading level, Darting does not have enough data traffic to embed routing updates in, handicapping the convergence rate. At 10% and 20%, enough traffic has become available to significantly reduce convergence times. Specifically, at 1% load, Darting takes 5.47 seconds to converge. At 10%, it takes 1.36 seconds, and at 20% it takes 1.86 seconds. This corresponds to a 75.77% decrease from 1% to 10%, and 51.40% increase from 10% to 20% load. (The higher number for 20% load is most likely due to those datapoints being results from a single iteration, while the 10% numbers are the average of 3 iterations.)

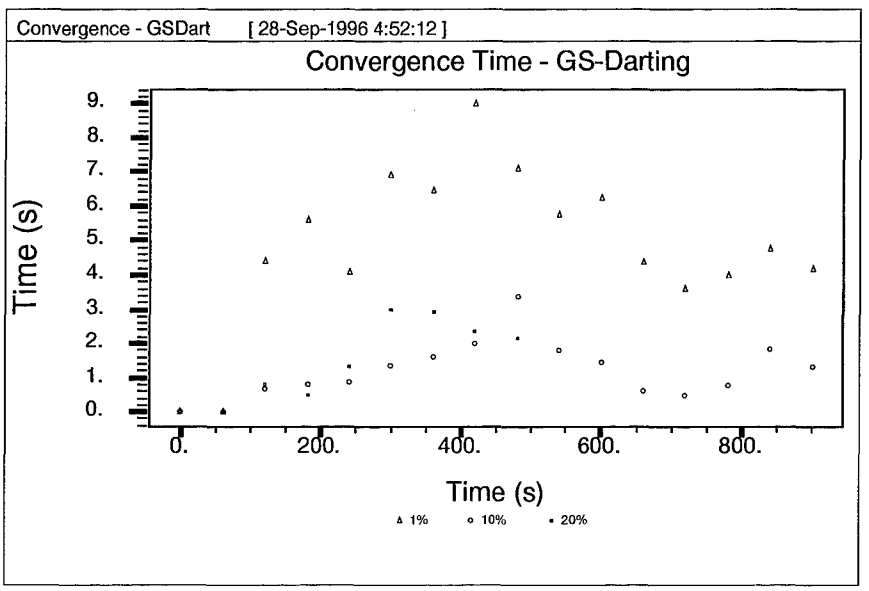

Figure 15: Darting Convergence Time in Globalstar

Darting overhead (Figure 16) also shows a strong dependency on the data rate, occupying a relatively constant percentage of the traffic. Overhead occupies 46.08, 44.27, and 44.13% of the total traffic at 1%, 10%, and 20% loads respectively.

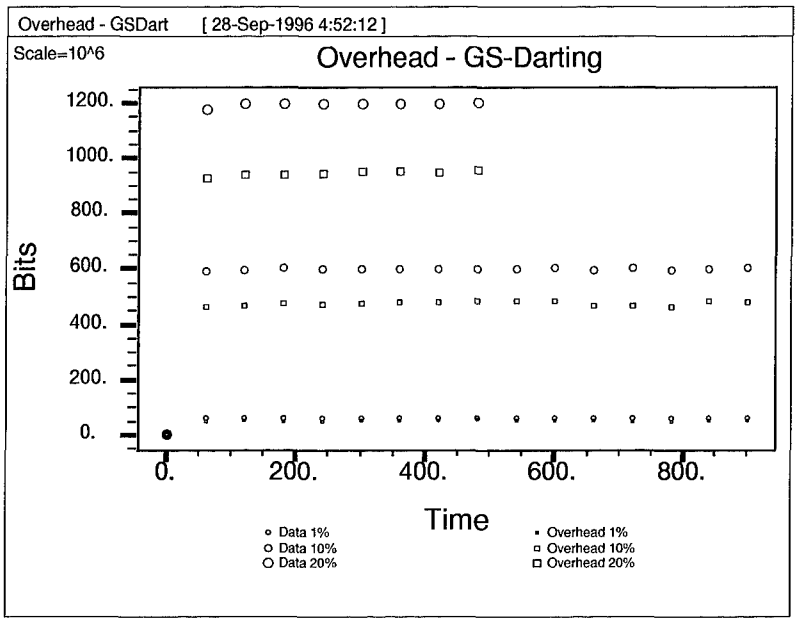

Figure 16: Darting Overhead in Globalstar

4.3.4 Darting (Iridium): Like Bellman-Ford, under Iridium Darting also displays a slight improvement in packet traversal times (Figure 17 and Figure 18). It has an average traversal delay of 0.0978, 0.1116, and 0.1087[9] seconds (13.72% and 14.40% changes) and an average maximum delay of 0.2645, 0.4179, and 0.3468 seconds (48.28% and 68.85%).

Convergence time (Figure 19) shows a drastic difference from Globalstar, with Darting being unable to handle the extra satellites during the periods of highest activity. Because of this, Darting averaged 12.29 seconds to converge at 1%, and failed to converge for some iterations during the positional update at 600 seconds. At this time there was insufficient data traffic for Darting to accommodate all the changes in the network. Darting averaged 7.99 seconds to converge at 10% and 1.067 seconds to converge at 20% load. (65.83% and 34.45% decreases.)

[9] The decrease in the 20% numbers is due to incomplete data. The simulations did not reach peak activity times.

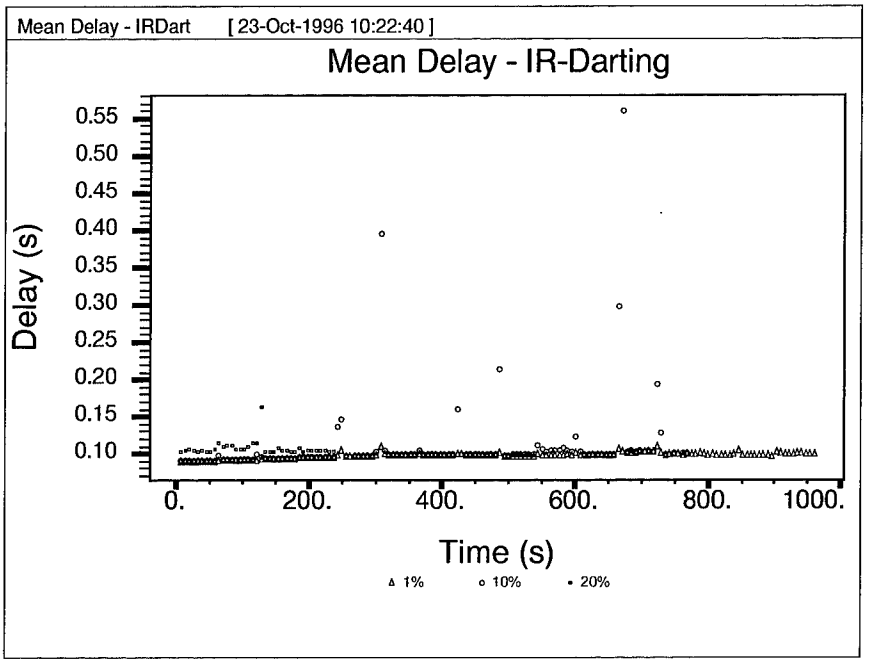

Figure 17: Darting Mean Delay in Iridium

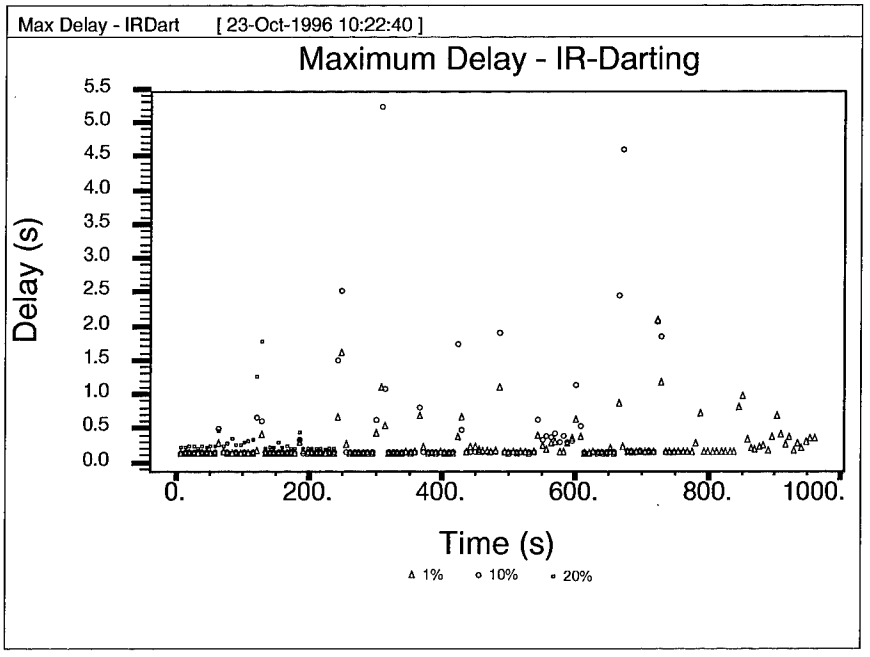

Figure 18: Darting Maximum Delay in Iridium

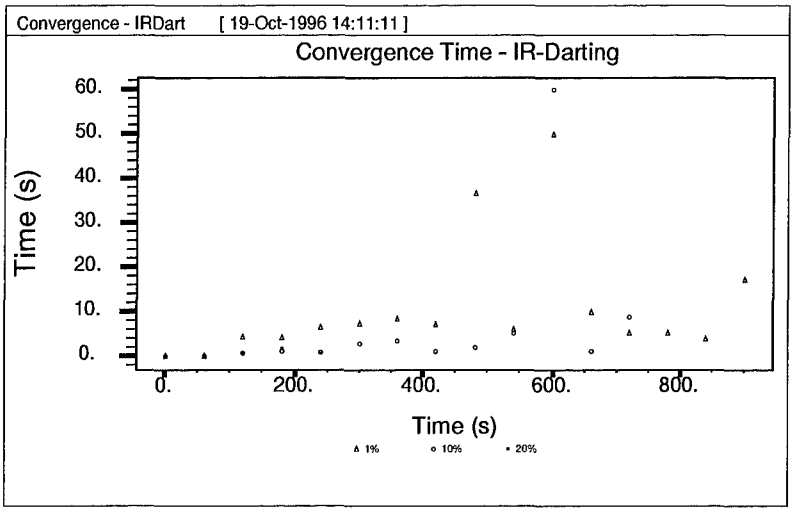

Figure 19: Darting Convergence Time in Iridium

Overhead (Figure 20) shares similar trends with the results from Globalstar, with the values showing a slight increase most likely due to the increased number of satellites. Overhead traffic accounted for 48.87%, 46.86%, and 45.42% of total traffic at the 1%, 10%, and 20% loads.

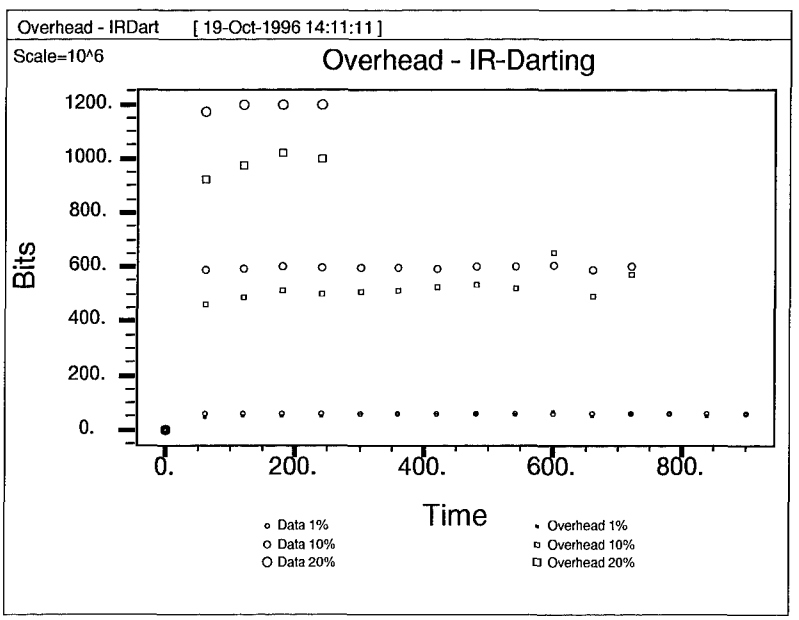

Figure 20: Darting Overhead in Iridium

4.4 Comparative Performance Results

4.4.1 *Mean Traversal Delay* Figure 21 and Figure 22 show close-ups of the mean delay performance. There is almost no significant difference between the traversal delay performance of the protocols on either constellation at any of the loading levels, except

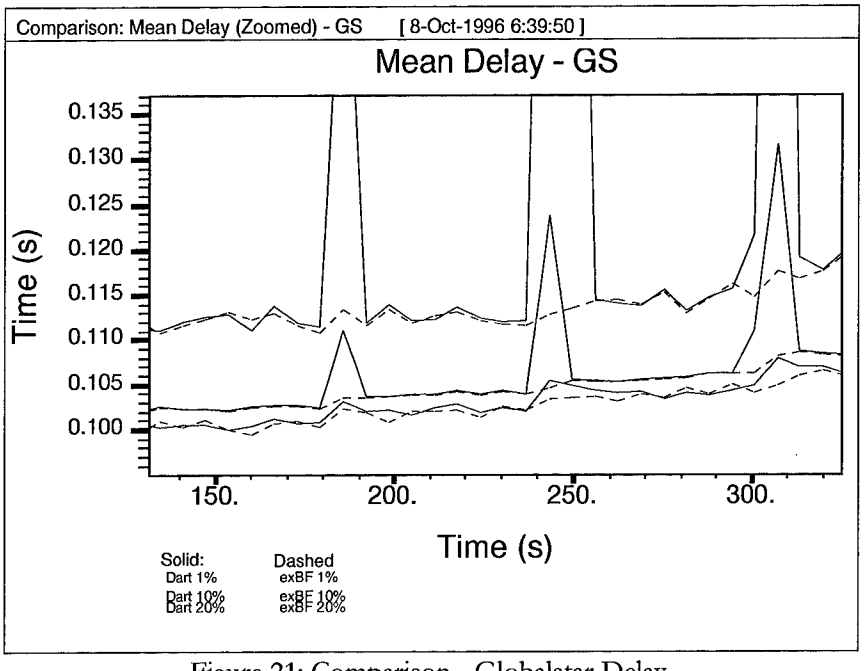

Figure 21: Comparison - Globalstar Delay

immediately after the a satellite positional update. At those times, Darting shows delays many times larger than those incurred by Bellman-Ford. In several cases, delays for Darting exceed the real-time threshold of 400ms by 200-300%. Over most of the range, Bellman-Ford enjoys fractionally better performance, most likely due to Darting's greater overhead (see below). Specifically, on average Bellman-Ford is 0.72% faster at 1% load, 2.39% faster at 10%, and 13.00% faster at 20% in the Globalstar constellation. It is 1.74% slower at 1%, and 7.56% and 3.074% faster at 10% and 20% in Iridium[10].

[10] The slower performance of exBF at 1% is due to the packet spike at 420 seconds mentioned in the previous section.

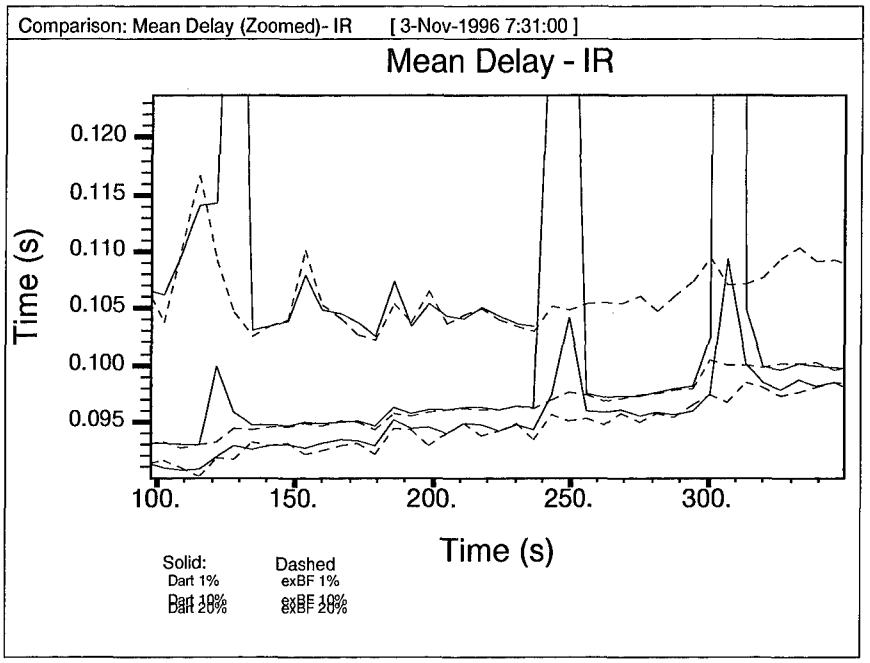

Figure 22: Comparison - Iridium Delay

Regarding convergence (Figure 24 and Figure 23), Darting's reliance on data traffic to piggy-back routing information makes it very sensitive to low data rates. At the loading levels investigated, Bellman-Ford turned in consistently better performance. At the lowest data rate, it was not uncommon for Darting to converge an order of magnitude slower than Bellman-Ford. The disparity narrows considerably at higher data rates though, and most likely becomes negligible at greater loads.

Under Globalstar, Bellman-Ford converged on average 3,582% faster than Darting at 1% load. It converged 764.4% and 1283% faster in the 10% and 20% cases. In Iridium, Bellman-Ford turned in performances 2661%, 496.2%, and 339.7% better than Darting.

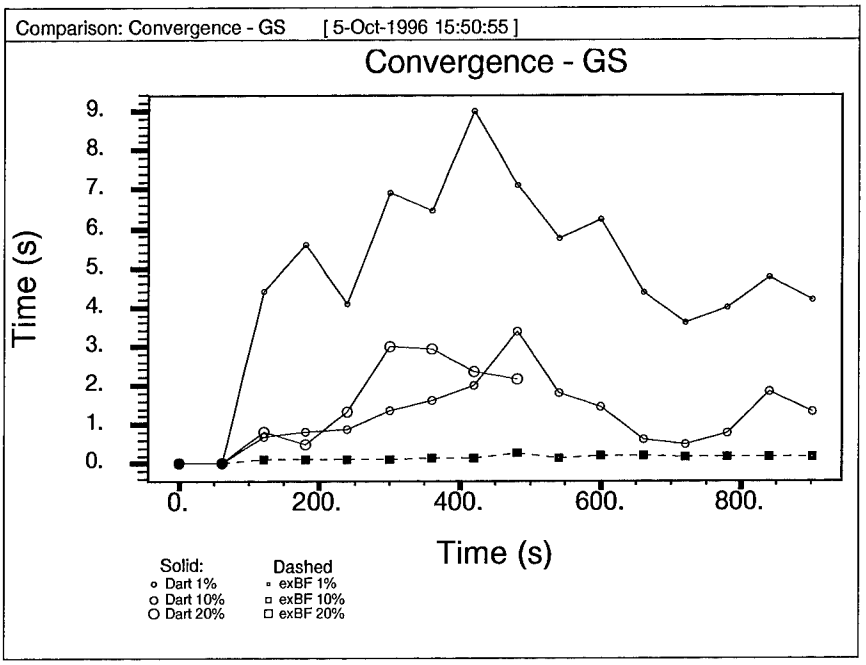

Figure 23: Comparison - Globalstar Convergence

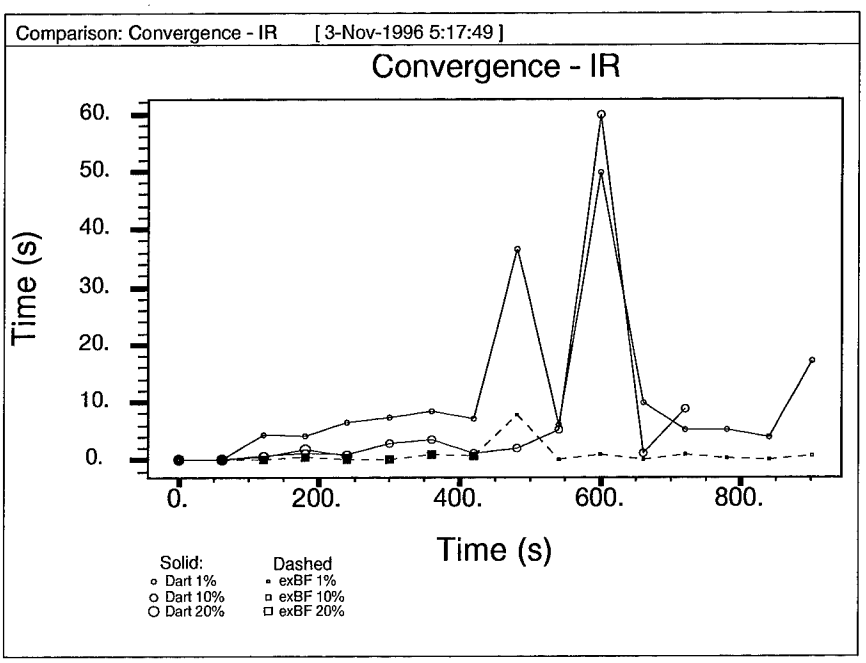

Figure 24: Comparison - Iridium Convergence

While the performance of both protocols is roughly equivalent for steady-state at higher data loads, Darting obtains this parity only at the expense of a much higher overhead (Figure 25 and Figure 26). In Globalstar, Darting has 149.1%, and 179.0% more overhead than Bellman-Ford at the 10%, and 20% levels. Only at the 1% loading level did Darting show better performance, having 15.38% less overhead. Under Iridium, Darting again has an edge only at the 1% data rate. However, while Darting enjoys an average 70.30% decrease in overhead from Bellman-Ford at 1%, at 10% and 20%, it incurs a 57.42% and 132.8% increase in overhead.

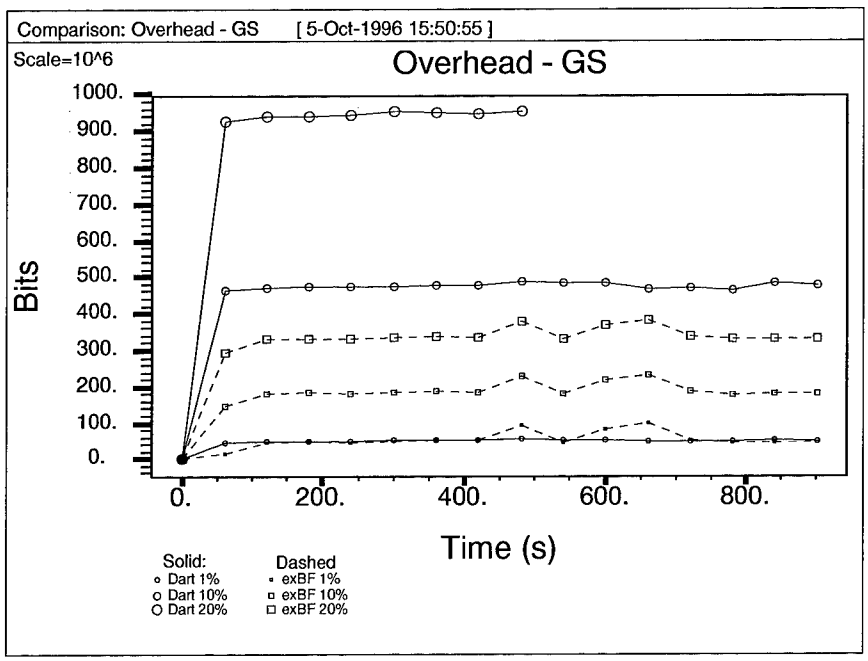

Figure 25: Comparison - Globalstar Overhead

Because Darting encodes link state information into every passing data packet, it is extremely sensitive to the resolution of this parameter. In this thesis, link state was encoded as a 32-bit integer corresponding to the inter-satellite distance[11]. Reducing this to a 16 bit

[11] The Designer built-in integer size is 32 bits.

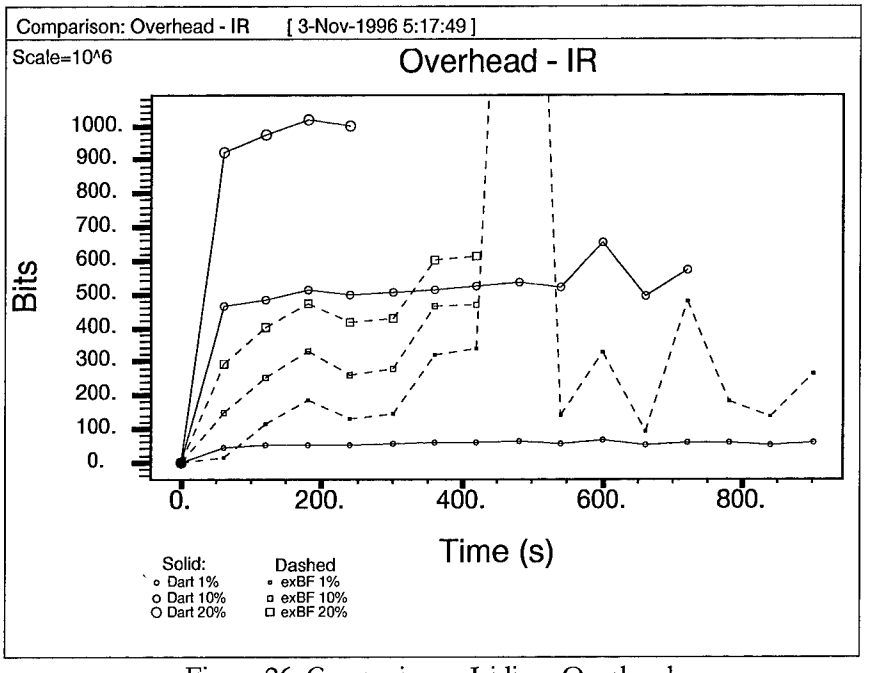

Figure 26: Comparison - Iridium Overhead

integer would improve Darting's performance roughly by a factor of two, bringing it closer to the Bellman-Ford numbers.

4.5 Summary

In almost every statistic measured, Extended Bellman-Ford had better performance than Darting. Darting's greatest handicap was the correlation between overhead, traffic intensity and path length. The requirement for each node to append its local environmental data to every passing packet resulted in a much higher overhead and slightly longer traversal times.

Darting also showed much higher traversal delay instability at network positional updates than Bellman-Ford. Again, this is most likely due to Darting's reliance on data-triggered convergence. Because propagation is the predominate delay component at these loading levels in the LEO environment, several packets from a burst may be forwarded incorrectly before updated routing information arrives at the transmitting node. Unfortunately, Darting will generate an update packet for *every* incorrect packet that is

transmitted, clogging up the reverse channel with unnecessary updates (which have queue priority over data packets.)

5. CONCLUSIONS

5.1 Protocol Applicability to the Orbital Environment

Contrary to expectations, for the characteristics measured there is a clear advantage to employing a distance vector routing protocol such as Extended Bellman-Ford over Darting as traffic intensities increase. Darting is severely handicapped by the requirement to place link data in each passing data packet. This results in overhead several times larger than Bellman-Ford on the same constellation, eliminating the savings realized from Darting's selective update mechanism. Massaging the frequency of update insertions and tweaking the resolution of the link weight function would seem critical to obtaining good performance from Darting. Better overhead results could have been obtained from Darting (at the expense of convergence rate) by re-adjusting these parameters.

Mean packet traversal delay for each protocol was within a few percentage points difference at all loading levels. However, neither protocol succeeded in keeping worst case delays completely within the 400 millisecond real-time limit. Maximum delays on the order of seconds were experienced by each protocol during some satellite positional update periods. Some of this delay may be an artifact of having satellite updates occur synchronously throughout the constellation. Allowing each satellite to initiate an update cycle independently would spread the update load over a wider time interval.

5.2 Simulation Problems Encountered

These simulations juggle a huge amount of data. Instantaneous RAM requirements during peak update periods for some of the simulations are more than 300MB. The 10% simulations take over a month of machine time to complete at top priority on a Sparc-20. If forced to use virtual RAM, the simulations complete less than 1 millisecond of simulation time per day.

Unfortunately, during the time the simulations were in execution, the half-life of the average Sparc station seemed to be about 2 weeks. Cumulatively, this caused the loss of close to a machine-year of simulation time. The addition of the ability to check-point a Designer simulation would have been invaluable. As it stands, further effort is needed to optimize the simulations for memory use before trying to extend this work.

5.3 Recommendations for Future Work

Aside from streamlining the memory requirements of the simulation, there are several theoretical aspects of the project that could be enhanced. First, the link formation subroutine of the routing protocols is embarrassingly primitive. Currently, links are formed with the closest 4 satellites, regardless of direction. Then those links are held until the partner satellite travels out of range, even if a more optimal satellite becomes available. Work should be done to form links at evenly spaced headings and optimal distances to yield better paths through the network.

Second, a link-state protocol (such as OSPF) should be added to the comparison to determine if any better performance can be provided by that class of algorithms. Also, the protocols should be exercised at higher loading levels if memory requirements can be lowered sufficiently to allow execution on the Sparc-20s.

Third, additional ground stations should be added. The single transmitter located in each octant of the globe did not yield a completely uniform load on each node in the network. Addition of more groundstations might yield smoother data. Again, this would only be feasible if memory requirements of the simulation were lowered.

5.4 Conclusion

The overriding delay component at the loading levels investigated is the propagation delay between nodes. Because there is usually only one optimal path to any destination, protocol merit is determined by how quickly and efficiently an algorithm can determine the components of that optimal path. In a satellite system, power is a critical resource. Thus

protocol overhead is an important parameter to manage. Each unnecessary bit of overhead is wasted transmission power. For the environments tested, Extended Bellman-Ford has a significant advantage over Darting in this area. It obtains equal performance faster and with a smaller overhead. More work needs to be done to optimize Darting before it should be considered for use in LEO networks.

APPENDIX A

Detailed Simulation Definition

A.1 Designer BDE

As briefly touched upon in Chapter 3, a top-down approach was used to design the simulation program files. This appendix will continue the discussion through all the Designer sub-schematics. Other than the actual routing blocks and the top-level schematics, all pieces of the simulation are identical for both protocols. The explanation will begin with the top-level memory and parameter variables.[12]

Starting at the top left of Figure 27, the six temporal parameters are fairly self-

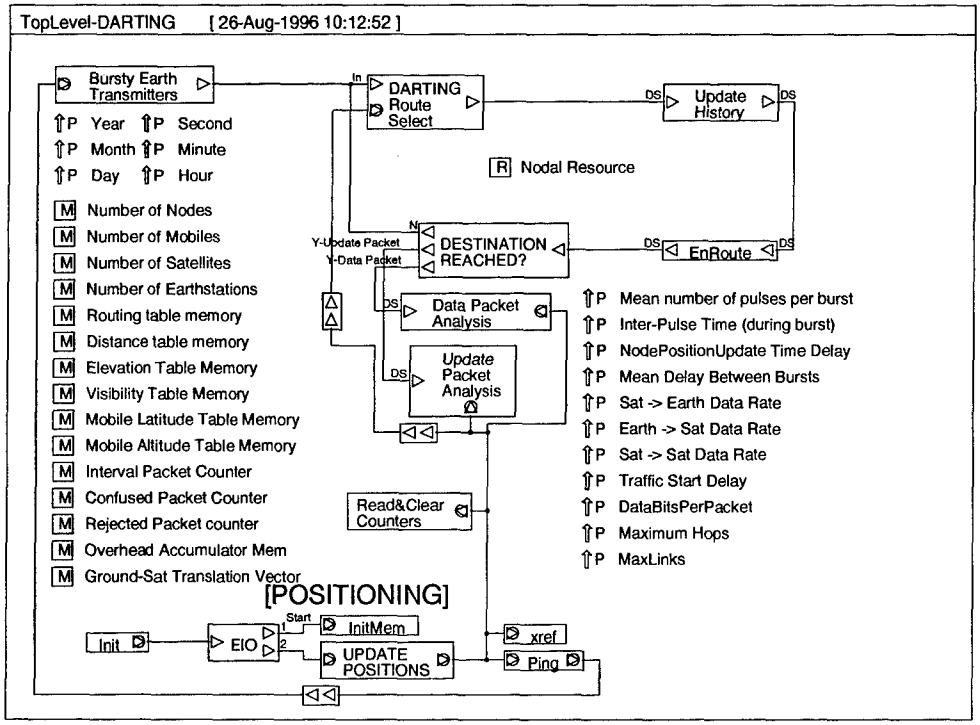

Figure 27: Simulation Top Level Schematic

[12] Throughout this appendix, italicized references represent entities created for this project, while quoted references represent Designer built-in constructs.

explanatory and are used to set the starting time of the simulation. For this simulation, these parameters were set at 0101:01 hours on 1/1/1998, due to the fact that Iridium is scheduled to become operational sometime in 1998.

The memory variable *Number of Nodes* is calculated in the InitMem block. It is the sum of the *Number of Mobiles*, *Number of Satellites*, and *Number of Earthstations* variables, which are passed into the simulation from the SatLab program based upon which of the two constellations is loaded. *Distance Table Memory, Elevation Table Memory, Visibility Table Memory,* and *Mobile Latitude and Altitude Table Memory* are also passed into the simulation by SatLab and reflect the current physical locations of the entities in the constellation. These variables are used by the routing protocol to calculate *Routing Table Memory*, which is a matrix of next-hops for every possible source and destination in the network.

The next four variables, *Interval Packet Counter, Confused Packet Counter, Rejected Packet Counter, and Overhead Accumulator Memory*, collect various statistics used by the analysis sections. Interval Packet Counter measures the total number of packets generated in the network during each satellite update period (60 seconds), Confused Packet Counter measures any packets that have exceeded the maximum number of hops (100), Rejected Packet Counter records the number of packets discarded due to insufficient queue space, and Overhead Accumulator Memory tallies the number of bits used for routing purposes during the update interval.

Finally, the *Ground-Sat Translation Vector* is used by the groundstations to look up the closest overhead satellite to use as a gateway into the network. It is calculated at the beginning of each update period by the xfer block.

The second column of parameters are set at runtime and determine the operational characteristics of the simulation. *Mean Number of Pulses per Burst, Inter-Pulse Time,* and *Data Bits per Packet* are arbitrarily set at 10 pulses, 1 microsecond, and 1024 bits. *Earth->Sat Data Rate, Sat-> Earth Data Rate,* and *Sat-Sat Data Rate* are set at 12.5 Mbps, 12.5 Mbps, and

25Mbps, based upon the data rates specified in the Motorola FCC filing for Iridium [FCC91]. From these, *Mean Delay Between Bursts* is calculated to provide an average data rate from each groundstation of 1, 10, 20 percent of the ground-space data rate. Higher data rates were infeasible due to RAM limitations on the Sun workstations, and the possibility of reducing the number of traffic sources was discarded due to Darting's sensitivity to non-uniform traffic distributions (See Appendix C.)

Node Position Update Time Delay holds the number of seconds between queries to SatLab for satellite position updates. As mentioned in Chapter Three, this parameter is set at 60 seconds.

Traffic Start Delay is used in the Ping block to delay start of the traffic generators to allow the initialization of the sub-blocks to complete. It is set at 1 second.

Maximum Hops is set at 100, and is used in the Update History block to remove any packet from the network that has visited more than 100 nodes in an attempt to reach its destination. Assuming an average packet in an optimal mesh network should have to travel no more than half the diameter of the network to reach any destination, 100 hops is slightly more than three times the radius of Iridium. Any packet that exceeds this limit is assumed to be trapped in a routing loop and is removed from the network.

Maxlinks is a parameter passed to the routing protocol to inform it of the maximum number of adjacent satellites that it may communicate with. Following Motorola's proposal for Iridium, this is set at 4 links.

A general overview of the operation of the top-level diagrams was included in Chapter Three (page 17).

The *InitMem* block (Figure 28) is used to query SatLab for the dimensions of the satellite constellation currently loaded. A type 1 request is sent into the BSIM primitive[13], which communicates with the SatLab program and retrieves the requested information. This is then stored in the memory variables as mentioned previously

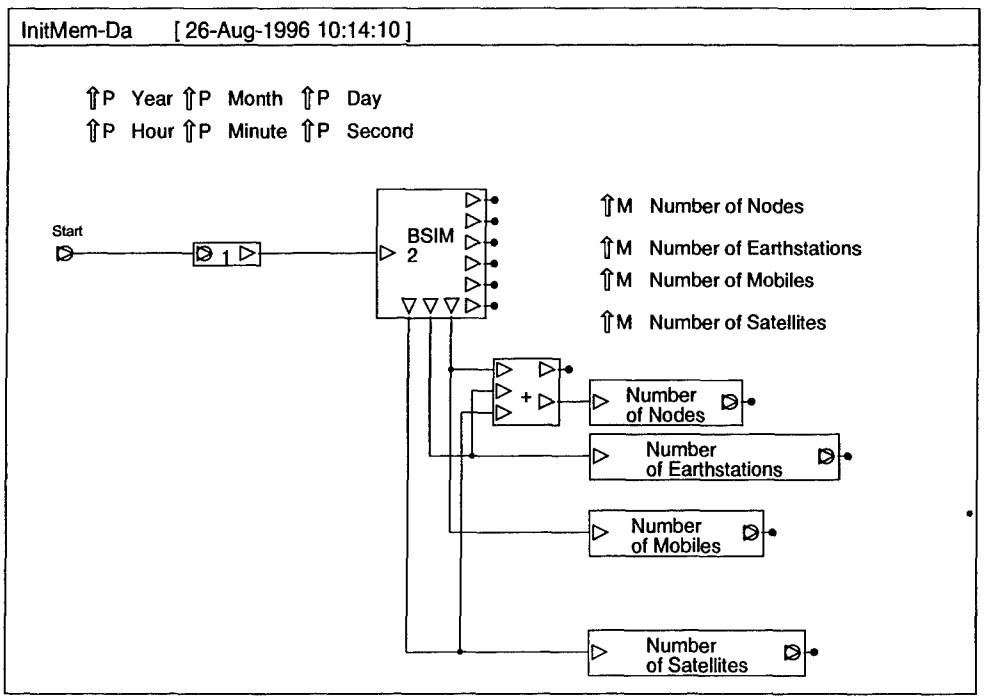

Figure 28: SatLab Memory Initialization

In the *Update Satellite Positions* block (Figure 29), the incoming trigger is used to query SatLab for the satellite positional information, and if any mobile users exist in the simulation, for their positions also. Additionally, the trigger is stored in a delay block set to the length of an update period as specified in the top-level diagram. The feedback loop around this delay block generates a new trigger at intervals equal to the specified delay. This has the effect of querying SatLab automatically at the end of every update period. These generated triggers

[13] A Designer "primitive" is a block whose contents are direct machine code. There are no schematics associated with a primitive.

are also passed back up to the top level diagram to allow action by the other blocks at the beginning of each new update period.

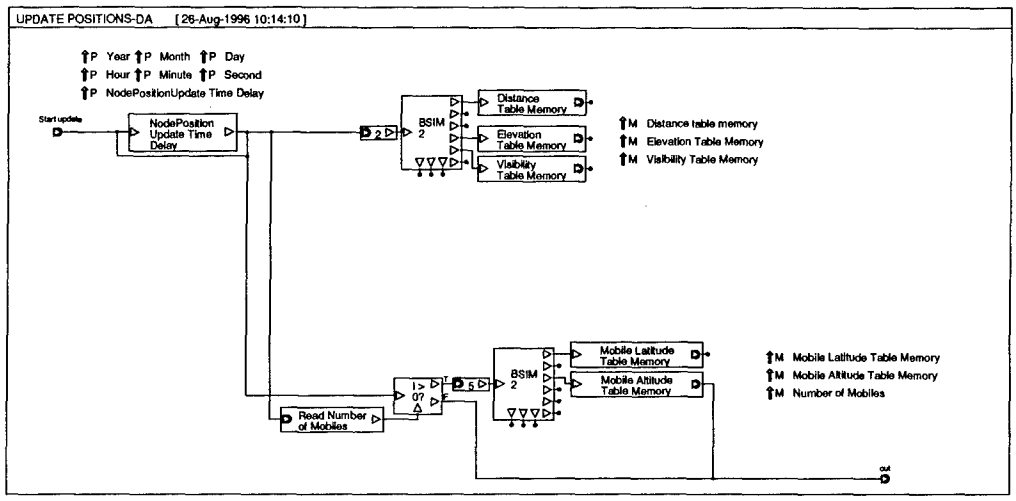

Figure 29: Update Satellite Positions Block

The purpose of the *Ping* block (Figure 30) is to allow only the first trigger of the simulation run through to initialize the traffic generators. As mentioned previously, this trigger is also delayed by a specified amount to allow the other blocks in the simulation to complete initialization before data packets begin to be generated.

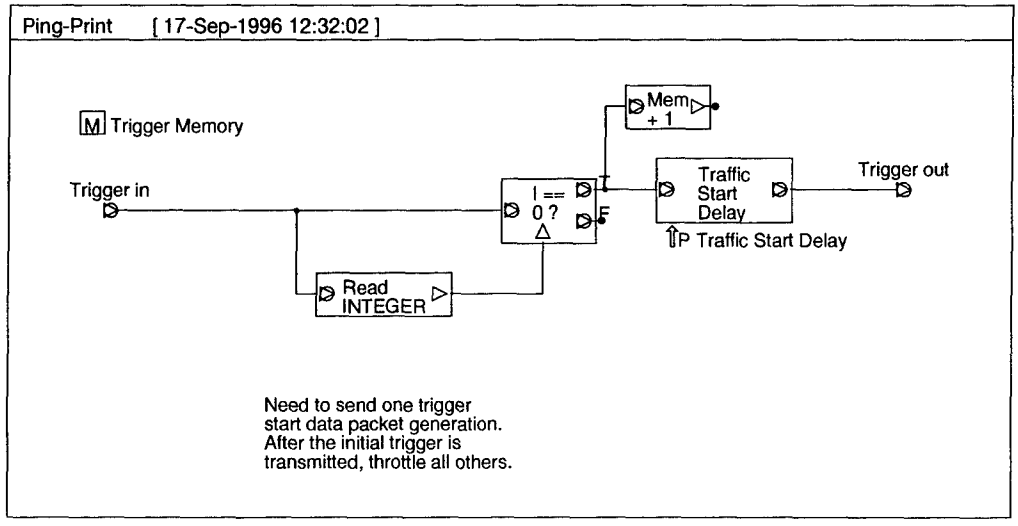

Figure 30: Update trigger "Ping" throttle

Xref (a sub-block in Figure 27) is a primitive block that calculates which satellite is nearest to each groundstation and updates the translation vector accordingly. As a primitive, there is no associated schematic for xref. Details of its implementation can be found in Appendix B.

The function of the *Read and Clear Counters* block (Figure 31) is fairly self-explanatory. Upon receiving the end-of-period trigger from *Update Positions*, the value of each of the specified memory variables is read and recorded for later analysis. Each variable is then reset to begin counting anew for the next period.

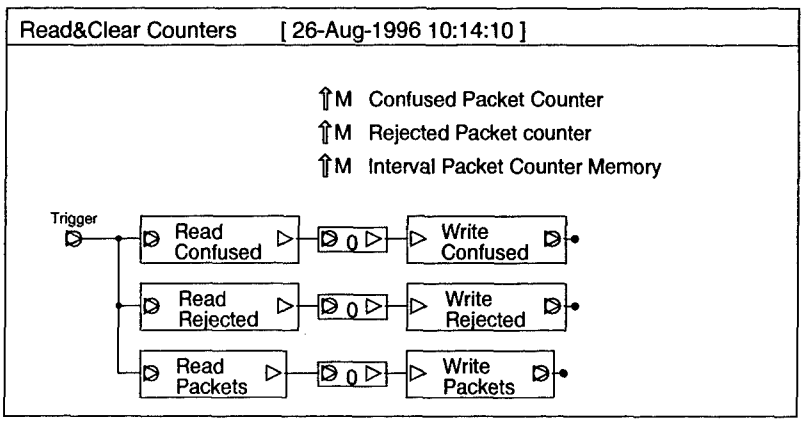

Figure 31: Read interval counters

Each of the eight groundstations in the simulation is represented by a *Bursty Groundstation* block (Figure 32.) Each groundstation has an independent "Bursty Source" traffic generator. This generator (slightly modified from a Designer built-in version) produces an

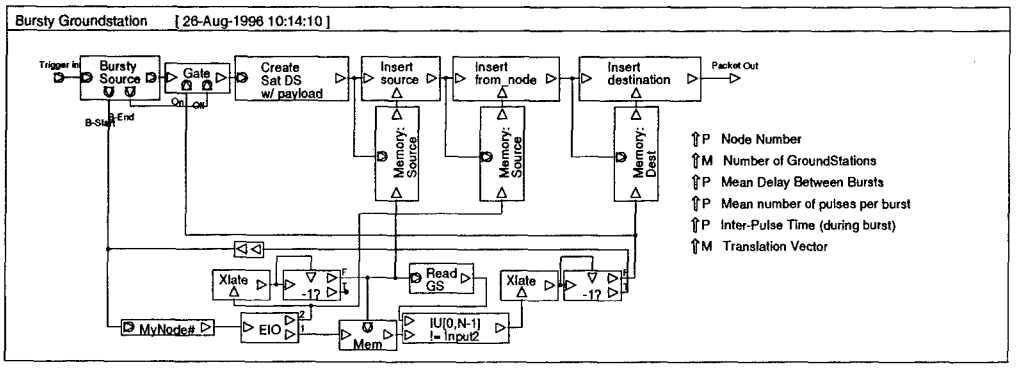

Figure 32: Bursty Groundstation Transmitter Block

exponentially distributed series of bursts, with individual burst sizes being geometrically distributed.

At the beginning of each burst, the source and destination of the burst are determined by the series of blocks along the bottom of the diagram. First, the node of the current node is sent into the EIO (Execute in Order) block. From here, it is loaded into two temporary local memories. After that, the value is passed to the "xlate" block, which is a built-in Designer primitive that reads the translation vector element corresponding to the current node. This produces the node number of a satellite, (or -1 if there is no satellite in range.) The node number of the source satellite is then stored in yet another local memory. If this groundstation does have a satellite in range, the number of groundstations is fed into a uniform random number generator to determine the destination for the burst. The random generator employed also takes an additional integer parameter that represents an illegal number for the output. By feeding in the current node, we can assure that we never generate a burst with the same source and destination.

Multiple Sources (Figure 33) holds an independent generator for each groundstation in the simulation.

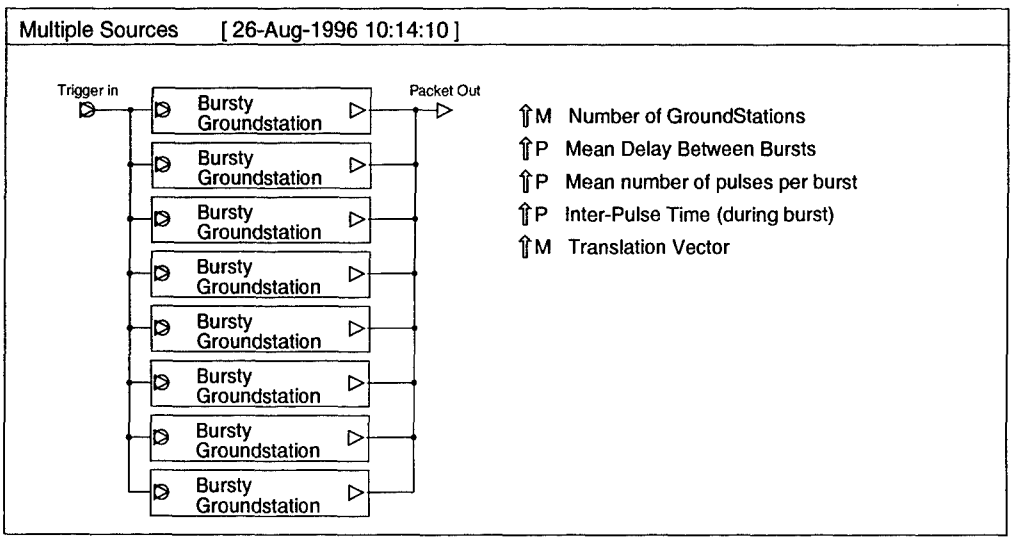

Figure 33: Transmitter Instances

Bursty Earth Transmitters (Figure 34) encapsulates the data generation function of all ground stations in the network. The incoming trigger signal initializes the multiple sources to begin packet generation. Each generated packet's length is set to the number of data bits specified in the corresponding parameter, plus 256 bits of overhead. (Eight header fields, each a 32 bit integer.) The cost field (used by Darting) is set to zero, and the packet type is set to one (signifying a new packet). Then a unique sequence number is added and the interval packet counter is incremented.

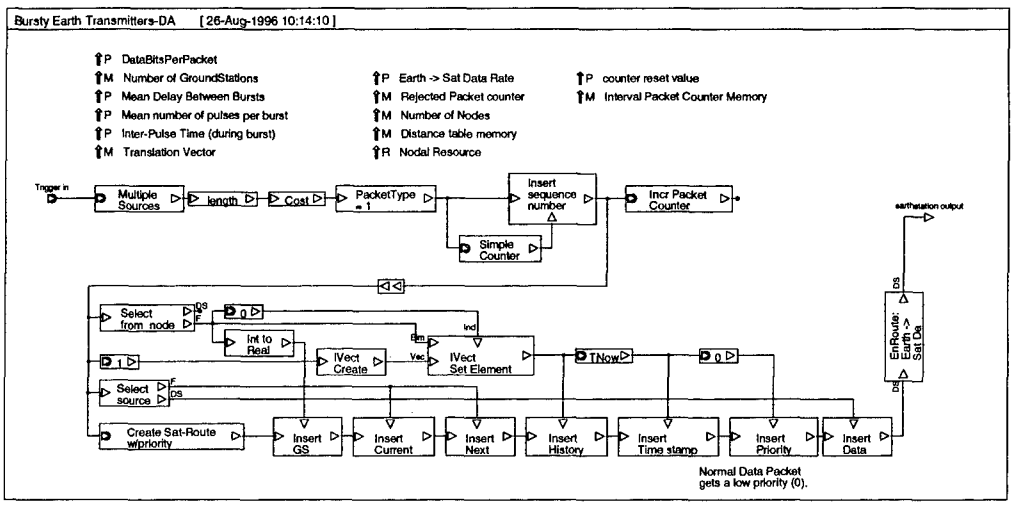

Figure 34: Bursty Earth Transmitters

The next section of the schematic encapsulates the data packet into a "Satellite Routing" data structure. This is a somewhat artificial construct introduced in the Designer satellite communication example, and allows a single instance of the routing primitive to service all satellites in the network. For reasons of compatibility with other concurrent thesis work, this functionality was retained, though some data is duplicated in the underlying data packets.

The from_node field is set by the traffic generator to its node id, and this value is placed into the Sat-Route packet for use by the transmitter block. A history list is generated and placed, along with the current time and correct priority, into the routing packet. Lastly the

calculated source satellite is placed in the current and next fields, and the data packet is encapsulated into the finished routing shell. From here, it enters the Earth to Space transmission block.

In the first stage of the *Earth-Sat* block (Figure 35), queuing at the outgoing buffer is simulated. The node number of the source is read from the incoming packet and fed into the dimensioned service block, which contains an independent queue for each node. A processing delay is added at this point, normally distributed with a mean of 100 microseconds and a variance of 5 microseconds [ClJ89]. The packet is timestamped upon entry and exit from the queue, and this information is used to update the accumulated delay recorded in the delay field. Any packets rejected are counted by the respective memory variable. The queue discipline is FIFO, with incoming higher-priority packets displacing low priority packets if the queue is full.

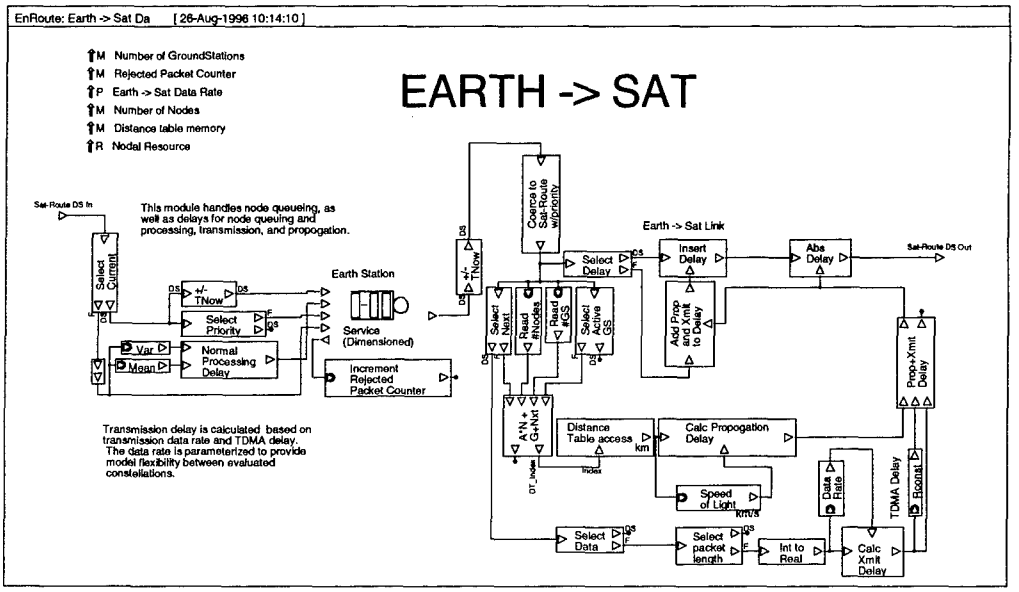

Figure 35: Uplink Delay (Earth-Sat) Block

At this point, propagation and transmission delays are calculated, recorded, and applied to the packet. First the index of the correct entry in the distance table is calculated based

upon the number of satellites and groundstations currently in the network. The distance is then used to calculate propagation delay, using a value of 2.97 x 10^8 for the speed of light. Simultaneously, the length of the packet is read and combined with the appropriate data rate to calculate the transmission delay. Additionally, an average TDMA access delay of 30ms is encountered, based upon Iridium's 60ms TDMA frame [FCC91]. At this point, the packet leaves the generation subsystem and appears on the top-level diagram.

Due to the fact that Bellman-Ford completely ignores packet data types, a kludge is required in the exBF top-level before a new packet enters the routing system, as shown below in Figure 36.

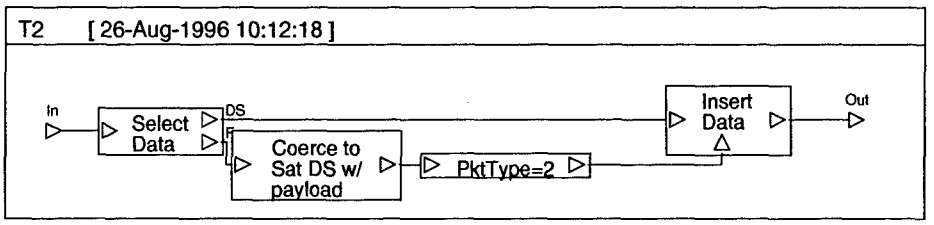

Figure 36: Bellman-Ford PacketType Kludge

Upon entry into the *Darting Router* (Figure 37), the data packet is removed from the routing shell. The shell then passes through a switch based upon the type of the data packet. Type 1 and 2 packets are data packets and will be rejoined with the routing shell after being passed through the Darting algorithm. Therefore the shells enter a simple FIFO buffer to

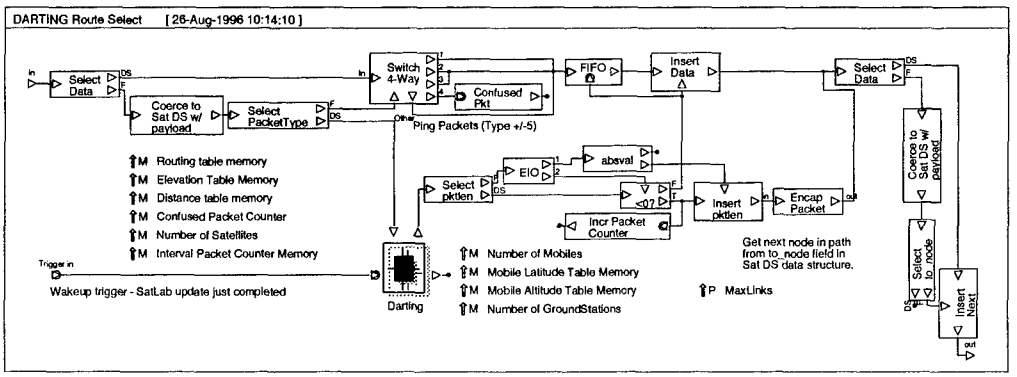

Figure 37: Darting Router

await their data packets. Type 3 and 5 packets are routing update packets, that will also be continuing along the network and thus must be rejoined with their shells. Type 4 packets are packets that are in error, or have reached their destination, thus should be removed from the network by the Destination Reached block. Thus, if a type 4 packet reaches the switch, it is an error, and the appropriate counter is incremented. Once the data packet has been updated with the correct next hop by the Darting algorithm and rejoined to its shell in the Insert Data block, the Next field of the routing shell is updated with the next hop calculated by Darting, and the re-combined packet exits the module.

Trigger pulses from the *Update Positions* block are passed to the Darting algorithm so that the network nodes may update their link topology tables as the satellites in the constellation move.

In the event that Darting detects an inconsistency in the current network topology after examining a data packet, it will generate additional routing update packets to correct the nodes that are in error. The Darting module signals the Designer framework that it has created an update packet by negating the packet length. Any packet that exits the Darting module with a negative packet length must have a routing shell created before it can proceed through the rest of the network. This is done in the *Encapsulate Packet* block, as shown below (Figure 38).

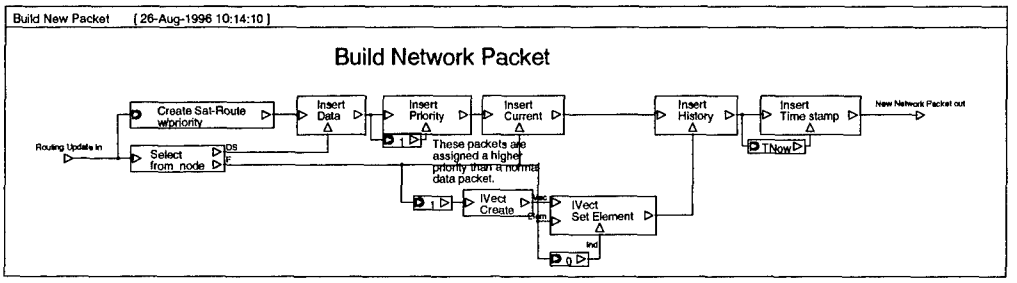

Figure 38: Encapsulate Packet

The Bellman-Ford routing schematic (Figure 39) is somewhat more complex due to the fact that the algorithm itself had no knowledge of packet types. It expects to receive only update packets, and never generates anything but update packets. Thus some additional "software" is required to interface the algorithm with the rest of the simulation.

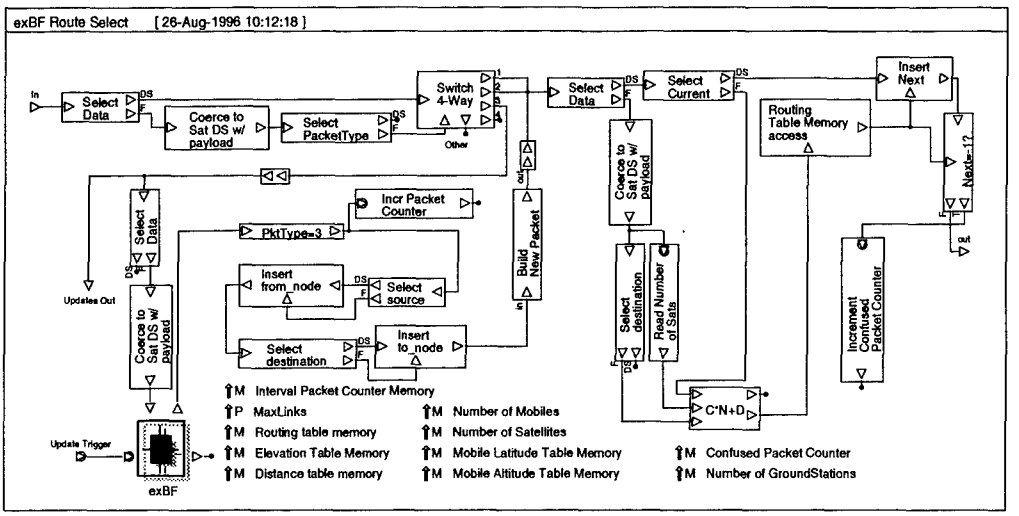

Figure 39: Extended Bellman-Ford router

As in Darting, the incoming data packet is removed from the routing shell and switched according to its packet type. Data packets are sent immediately on, while update (type 3) packets are sent into the router. Just as in Darting, outgoing update packets must have a routing shell built for them before they continue.

Triggers from the *Update Positions* block are passed to the exBF algorithm also so it can update its topology tables.

Because the exBF router never handles data packets itself, but only maintains a table of next-hops, the correct value for the Next field in the routing shell must be read and inserted in the shell before the packet leaves the router. If the current node knows of no path to the destination (signified by a -1 in the next hop table), it increments the confused packet counter and discards the packet.

The main functions of the *Update Packet History* block (Figure 40) are to remove packets that have exceeded the maximum hop count from the network, and to maintain the history vector of a packet as it passes through the network.

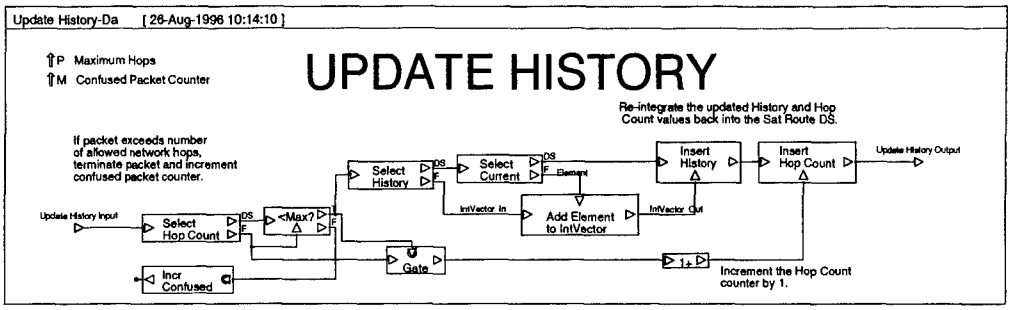

Figure 40: Update Packet History

The *EnRoute* block's function (Figure 41) is solely to shuttle type 4 packets (which have reached their destination) around the Sat-Sat queuing block. This prevents the packets from picking up spurious delay by passing through the final link twice.

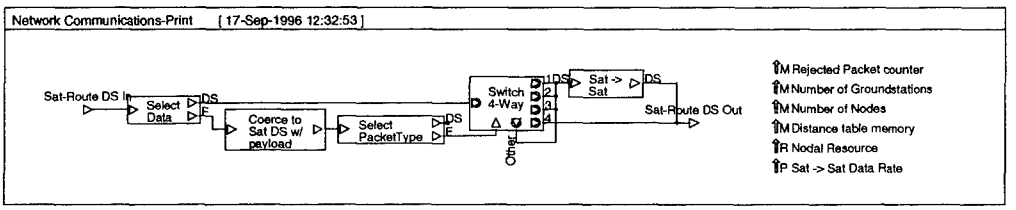

Figure 41: EnRoute in Space

The *Sat-Sat Delay* block (Figure 43) is identical in function to the Earth-Sat delay block, save a slight modification in the index calculation and the absence of TDMA delay.

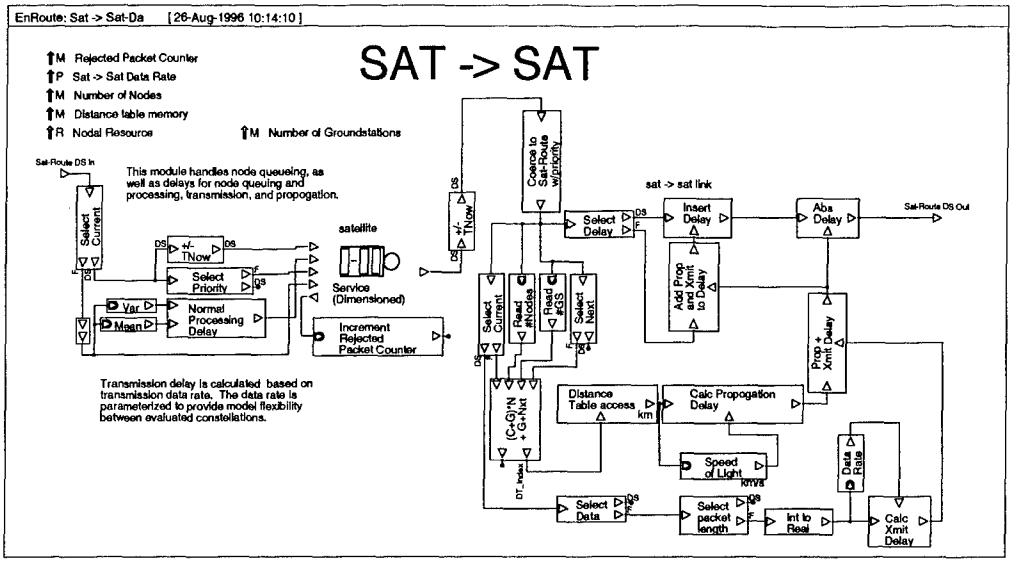

Figure 43: Satellite to Satellite Delay Block

The function of the *Destination Reached* block (Figure 42) is to check to see if a data packet has reached the satellite above its destination. (Thus type 3 and 5 update packets are passed immediately through.) It is also here, once the packet has passed through the propagation delay in the Sat-Sat block, that the Current node field is updated to reflect the packet's new position in the network.

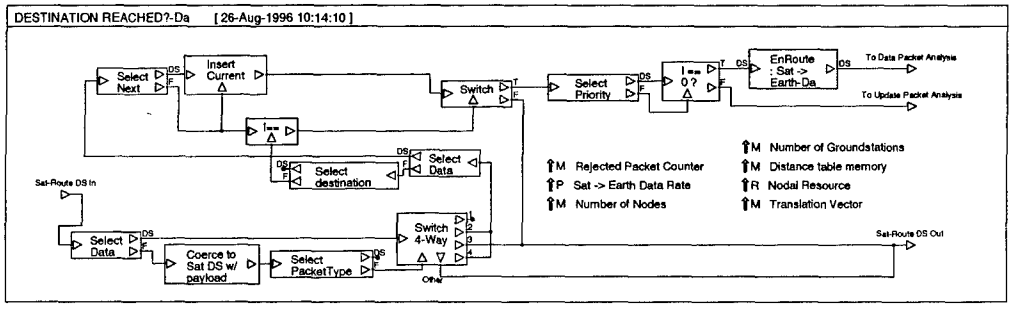

Figure 42: Destination Reached Block

If the destination is not equal to the current node, the packet is passed back to the routing node to begin an other cycle. If it *is* at its destination, the packet is passed to the

appropriate analysis block for counting purposes; first passing through a downlink if the packet is destined for the surface.

The *Sat-Earth Delay* block (Figure 44) is similar to the other two delay blocks, save for the necessity to perform a translation vector lookup to determine the node number of the groundstation currently being serviced by this satellite.

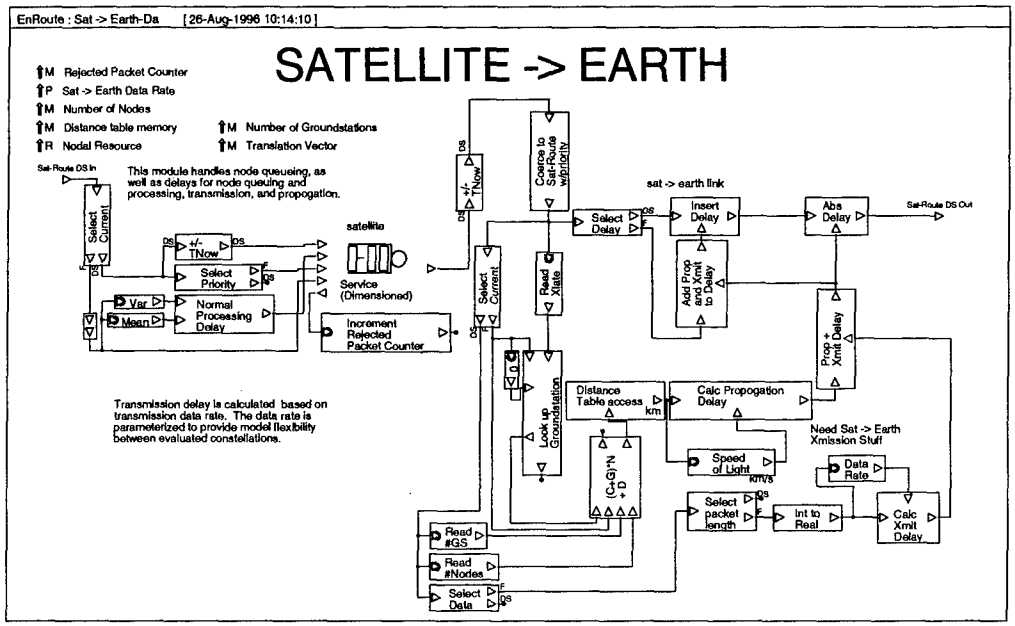

Figure 44: Sat - Earth Delay Block

Once a data packet reaches its destination, the delays and overheads associated with its network traversal are recorded in the *Data Packet Analysis* block (Figure 45). Delay is recorded directly by a Designer probe on the incoming routing shell's delay field (not

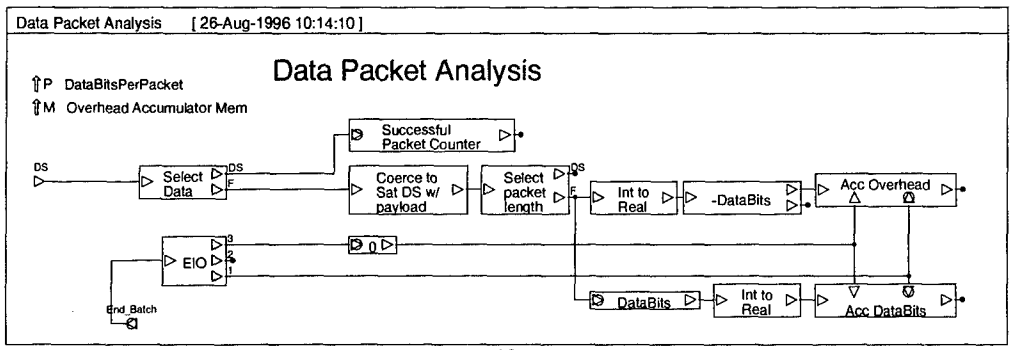

Figure 45: Data Packet Analysis

shown.) Overhead, however, is tallied on an update-by-update basis using accumulators that are read and cleared by triggers from the Update Positions block. Overhead is calculated by subtracting the number of databits in the packet from the total number of bits the packet has accumulated on its trip through the network. The databits accumulator is simply incremented by the number of bits in a packet every time a packet arrives.

Similar to the *Data Packet Analysis* block, the *Update Packet Analysis* block (Figure 46) records the overhead imposed on the network by the update packets. In this case however, it is somewhat easier as the entire packet constitutes overhead.

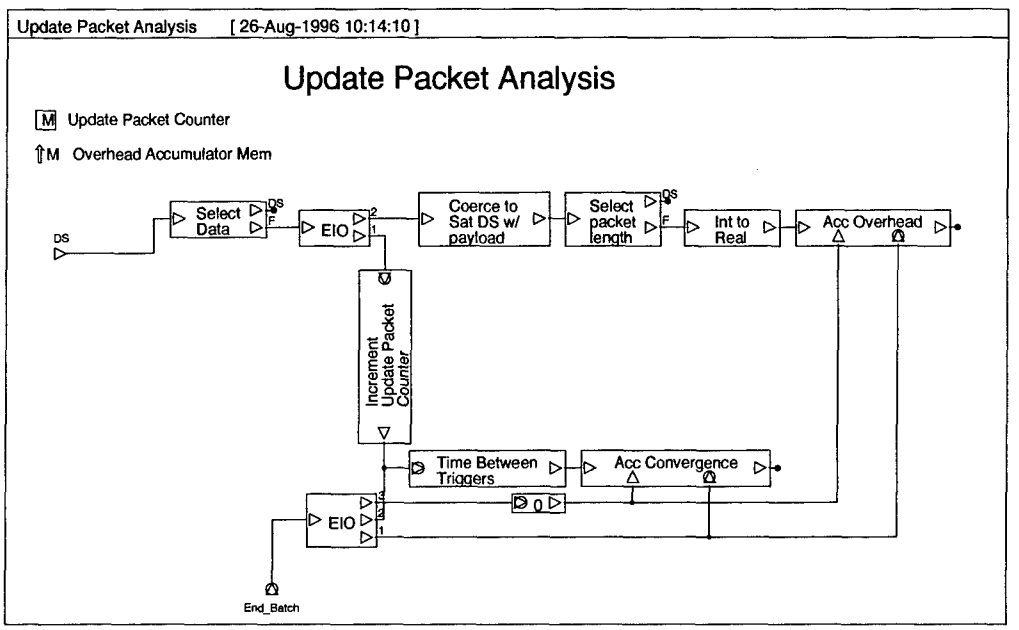

Figure 46: Update Packet Analysis Block

Additionally, the block is responsible for keeping track of the amount of time required for the routing protocol to converge. As was mentioned in Chapter 3, this is accomplished by measuring and accumulating the amount of time between every delivered update packet, based upon the premise that when there are no more update packets circulating the network, the protocol has converged as far as it can. This accumulator is also read and cleared by a trigger pulse from the Update Positions Block.

A.2 Packet Format

As alluded to in the preceding discussion, two different packet data structures are used in the simulation. They are defined as follows:

Table 2: Sat Route w/ priority data structure

Field	Data Type	Range	Default Value	Inherited From
Current	INTEGER	[0,+Infinity)	...	Sat-Route DS [SatCom_dbs]
Next	INTEGER	[0,+Infinity)	...	Sat-Route DS [SatCom_dbs]
Data	Sat DS	...	...	Sat-Route DS [SatCom_dbs]
Time stamp	REAL	[0,+Infinity)	...	Sat-Route DS [SatCom_dbs]
BitErrorRate	REAL	[0,+Infinity)	0.0	Sat-Route DS [SatCom_dbs]
EbN0	REAL	[0,+Infinity)	0.0	Sat-Route DS [SatCom_dbs]
Delay	REAL	(-Infinity, +Infinity)	0.0	Sat-Route DS [SatCom_dbs]
Hop Count	INTEGER	[0,+Infinity)	0	
History	INT-VECTOR	...	...	
Priority	INTEGER	[0,+Infinity)	0	

Table 3: Sat DS w/ payload Data Structure

Field	Data Type	Range	Default Value	Inherited From
source	INTEGER	[0,+Infinity)	...	Sat DS [SatCom_dbs]
destination	INTEGER	[0,+Infinity)	...	Sat DS [SatCom_dbs]
packet length	INTEGER	[0,+Infinity)	...	Sat DS [SatCom_dbs]
sequence number	INTEGER	[0,+Infinity)	...	Sat DS [SatCom_dbs]
PacketType	INTEGER	(-Infinity, +Infinity)	1	
Cost	INTEGER	(-Infinity, +Infinity)	0	
from_node	INTEGER	(-Infinity, +Infinity)	-1	
to_node	INTEGER	(-Infinity, +Infinity)	-1	
Payload	INT-VECTOR	...	...	
scl_list	VECTOR	...	...	

APPENDIX B

Custom Designer Primitives

Several functions necessary for building the simulation could not be easily created with the primitives provided in Designer. To alleviate this problem, it is possible to write custom primitives in C++ to provide additional functionality to Designer. The primitives that were created for this simulation were the two routing protocols, the cross-reference utility "xref", and one vector support block. The details of these custom primitive blocks, along with their C++ code, can be found below. The discussion assumes familiarity with C++.

Note that Designer automatically generates a C++ shell for custom primitives that handles all the interfacing with the simulation engine. Thus, the portions of the programs that are machine generated will not be displayed or discussed. For more details on those portions, see the Designer Modeling Reference Guide [Alt94]

B.1 Darting

The Darting algorithm is the easier of the two to understand, as it is an almost exact implementation of the algorithm as detailed by Tsai and Ma [TsM95]. One extension to the protocol had to be made to accommodate the granularity of the traffic sources in the simulation. Details of the problem and the implemented solution can be found in Appendix C.

The packet types of Darting are differentiated by a packet type field in each packet. Type 1 packets are new data generated by one of the ground stations. Type 2 packets are data packets that have already been processed through at least one node. Type 3 packets are predecessor update packets as described in [TsM95]. Type 4 packets are packets that have reached their destinations or are in error that need to be removed from the network and

processed in the analysis blocks. Type 5 packets are "ping" packets that implement the Darting extension mentioned above.

o External Ports:

Input:

SyncUpdate is of type TRIGGER: Description: Triggers bulk link cost updates using the latest data from SatLab. The primitive will iterate through every node and generate any necessary updates.

inbound_update is of type Sat DS w/ payload: Description: Accepts input Sat DS w/ payload packets.

Output:

outbound_update is of type Sat DS w/ payload: Description: Outputs Sat DS w/ payload packets with appropriate topology update messages based upon the input changes. One input may result in many several output packets.

routematrix is of type INT-VECTOR: Description: Outputs a Satcom_dbs style global route table based upon any input changes. However, the primitive updates the global memory directly, so this output is fairly useless, and is only included for backward compatibility with Designer.

o External Arguments: (For details, see Appendix A.)

(M) "Routing table memory" is of type "INT-VECTOR"

(M) "Elevation Table Memory" is of type "REAL-VECTOR"

(M) "Distance table memory" is of type "REAL-VECTOR"

(M) "Number of Satellites" is of type "INTEGER"

(M) "Number of Mobiles" is of type "INTEGER"

(M) "Mobile Latitude Table Memory" is of type "REAL-VECTOR"

(M) "Mobile Altitude Table Memory" is of type "REAL-VECTOR"

(M) "Number of GroundStations" is of type "INTEGER"

(P) "MaxLinks" is of type "INTEGER"

o Internal Arguments: These arguments hold the local data of each router.

(M) "DummyList" is of type "LIST"

Initialization Value: Uninitialized

Description: Just included so Designer will include the Linked List header file automatically.

(M) "RouteDistance" is of type "REAL-VECTOR"

Description: Holds each node's estimated distance to each destination.

Initialization Value: Vector length: (1) Initial Value: 0.0

(M) "NodeMemory" is of type "VECTOR"

Description: Stores routing state from iteration to iteration.

Initialization Value: Vector length: (1) Initial Value: Uninitialized

(M) "OutputQueue" is of type "VECTOR"

Description: Holds packets awaiting output from each node.

Initialization Value: Vector length: (1) Initial Value: Uninitialized

(E) "NextPacket" is of type "EVENT-LIST"

Initialization Value: Uninitialized

Description: Used to schedule a simulation event to output packets from OutputQueue.

(M) "Neighbors" is of type "INT-VECTOR"

Description: Holds the state of the network links.

Initialization Value: Vector length: (1) Initial Value: 0

```
/*
 *        Module Name :                 Darting
 *        Template Created By :         3.0
 *        Author:                       rjanoso
 *        Last Modification Date:       29-Jul-1996 12:16:29
 *        Template Date:                29-Jul-1996 12:16:42
 */
```

The first user-defined section of the Designer template holds global includes and defines that apply throughout the class. In this instance, ADJLIST is a macro used simplify casting later in the code. Because Darting does not specify how least-cost paths to a destination are to be determined, I chose to implement a Dijkstra subroutine to fill that need. This specific Dijkstra implementation [CoL90] uses an adjacency list format for the network graph. Each network node has a vector of these lists, one for every other node in the network. ADJLIST(X,Y) accesses node X's list of all the outgoing edges from node Y. LIST_t and VECTOR_t are built-in Designer classes for linked lists and arrays.

SATDIST is another macro used to abstract the index calculation into the vector of distances passed into the simulation from SatLab. SatLab encodes each row of the distance

table matrix into one long vector, beginning with the groundstations. SATDIST returns this to a matrix-type format, while automatically adjusting for extraneous groundstation data.

PACKETHEADERSIZE is used in packet length calculations, and is based upon eight 32-bit fields in the packet header.

```
/****  Includes and Defines Below Here ****/
#define  ADJLIST(X,Y) ((LIST_t&)(((VECTOR_t&)Adjacency[X])[Y]))
#define  SATDIST(X,Y) Distancetablememory[gstations*(gstations+nodes) +
X*(gstations+nodes)+(gstations+Y)]
#define PACKETHEADERSIZE 256
/****  Includes and Defines Above Here ****/
```

This section of the Designer Template is used to create global functions and variables that can be referenced throughout the class. Detailed descriptions of the functions can be found preceding the actual code.

```
/****  Instance Definitions Below Here ****/
  void Init();
  void CheckInit();
  void update_links();
  void rt_update(int curr_node);
  void Return_Ping();
  void Build_Initial_AdjLists();
  void Check_Degenerate_Type3(int n_curr, int dest, int n_via, int p_cost);

  inline void SyncLinks(int curr_node, int scln_a);
  inline void SyncLink (int curr_node, int scln_a, int scln_b);
  inline void Relax(int n_curr, int n_from, int n_to);

  int FindNode(int node, LIST_t& AdjList);

  INTEGER_t tempint;
  INTVECTOR_t tempiv;
  TRIGGER_t trig;

  LIST_t OutList;
  VECTOR_t local_scl(1);
  VECTOR_t Adjacency(1);
  VECTOR_t EmptyAdj;
  SatDSwdpayload_t *TempPacket;
  INTVECTOR_t RouteMatrix,outvect, Vi, payload;
  REALVECTOR_t DistMatrix;

  double simtime;
  int gstations,seqnum,outindex;
  int nodes;
  int MAXLINKS;
  int   *neighbor;
  int  **NeighMatrix;
  int  **RDist;
  int  **Pred;
```

```
  const float MAXDIST = 1000000000.0; //Satlab uses ten billion for infinity, thus
1e+9 is safe.

  int initflag;
/****  Instance Definitions Above Here ****/
```

The next section can be used to initialize the global variables:

```
/****  User Constructor Code Below Here ****/
initflag = -1;
outindex = 1;
/****  User Constructor Code Above Here ****/
```

The bulk of the working code appears in this section:

```
/****  User Code Below Here ****/
```

Init is a special function included in the Designer template that is called once for each primitive during instantiation. Here I use it to set up the event scheduling for output packets.

```
void Darting::Init()
{
  EVENTLIST_t Temp = NextPacket;
  Temp.Extend(NextPacket_Entry());
  SetNextPacket(Temp);
}
```

CheckInit is another initialization function that I included to do dynamic arrays based upon the size of the constellation being simulated. Because the size of the constellation is unknown until it is read from SatLab, this setup cannot be done at instantiation. *Neighbor* is used to extract the local neighboring nodes from *NeighMatrix*, which holds the link information for the whole network. *RDist* holds the least cost distances from the current node to all other destinations, and *Pred* is the predecessor matrix associated with those distances. *Adjacency* is a matrix[14] of linked lists, which holds each node's view of the network

[14] Actually, a vector of vectors.

for use by the Dijkstra algorithm, while *local_scl* holds each node's locally detected status changed links.

```
void Darting::CheckInit()   //Initialize global variables if needed
{
  int i,k;
  if (initflag == -1) {
    initflag = 0;

    neighbor = new int[MAXLINKS];
    if (neighbor == 0) {cerr << "Out of Memory!";}

    NeighMatrix = new int*[nodes];
    if (NeighMatrix == 0) {cerr << "Out of Memory!";}
    for (i=0;i<nodes;i++) {
      NeighMatrix[i] = new int[MAXLINKS];
      if (NeighMatrix[i] == 0) {cerr << "Out of Memory!";}
      for(k=0;k<MAXLINKS;k++) NeighMatrix[i][k] = -1;
      }//for i

    RDist = new int*[nodes];
    if (RDist == 0) {cerr << "Out of Memory!";}
    for (i=0;i<nodes;i++) {
      RDist[i] = new int[nodes];
      if (RDist[i] == 0) {cerr << "Out of Memory!";}
      for(k=0;k<nodes;k++) RDist[i][k] = -1;
      }//for i

    Pred = new int*[nodes];
    if (Pred == 0) {cerr << "Out of Memory!";}
    for (i=0;i<nodes;i++) {
      Pred[i] = new int[nodes];
      if (Pred[i] == 0) {cerr << "Out of Memory!";}
      for(k=0;k<MAXLINKS;k++) Pred[i][k] = -1;
      }//for i

    LIST_t AdjList;
    VECTOR_t AdjVect(nodes,AdjList);
    EmptyAdj = AdjVect;
    Adjacency.ChangeLength(nodes);
    for (i=0;i<nodes;i++) Adjacency[i] = AdjVect;

    DistMatrix.ChangeLength(nodes*nodes, MAXDIST);
    RouteMatrix.ChangeLength(nodes*nodes,-1);
    tempiv.ChangeLength(1,-1);
    local_scl.ChangeLengthVECTORDefaultValue(nodes,tempiv);
    local_scl[0]=tempiv;

    }//if initflag
}//CheckInit
```

The FindNode function determines if the specified destination node is an immediate neighbor to the current node. If the requested destination is adjacent to us, it returns the index into the adjacency list of the link information. If not, it returns -1. The Designer linked lists are prioritized lists, and the node number of the destination is encoded as the priority of the list element. Thus, each linked list is always kept sorted by node number,

simplifying the search problem. The actual data kept in the list is the cost associated with traversing the link.

```
//Function to find the desired node in an Adjacency List. Returns -1 if node is not in
list.
int Darting::FindNode(int node, LIST_t& AdjList)
{
  int i=0;
  unsigned int priority=0;
  double time_entered;
  for (i=0;i<AdjList.Length();i++) {
    AdjList.GetElm(i,priority,time_entered);
    if (priority == node) return(i);
  }//for i
  return(-1);
}//FindNode
```

Each node in the network has an entry in its adjacency tables for every other node. This reflects the current node's view of the network. However, because each link is bi-directional, it is actually entered as two entries in the adjacency "matrix". The following functions make sure that if we change the information for one direction of the link, the reverse direction is also updated.

```
//Synchronizes link b->a with link a->b.
//If a->b does not exist, and b->a does, b->a will be deleted.
//If a->b exists and b->a doesn't, b->a will be created.
//Otherwise the value of b->a will be made equal to a->b.
inline void Darting::SyncLink(int curr_node, int scln_a, int scln_b)
{
  LIST_t& Adj_a = ADJLIST(curr_node,scln_a);
  LIST_t& Adj_b = ADJLIST(curr_node,scln_b);
  LIST_t::QueueOrderings FIFO = (LIST_t::QueueOrderings)0;
  int x=0,y=0;

  x=FindNode(scln_b,Adj_a);
  y=FindNode(scln_a,Adj_b);

  if (x == -1 && y != -1) delete(Adj_b.Remove(y));
  if (x != -1 && y == -1) Adj_b.Enqueue(Adj_a[x],FIFO,scln_a);
  if (x != -1 && y != -1) Adj_b[y] = Adj_a[x];
}//SyncLink

//Syncs all possible links at the specified node, except updates
// to our local links are not allowed. (Protocol can get confused.)
inline void Darting::SyncLinks(int curr_node, int scln_a)
{
  int i;
  unsigned int n;
  double etime;
  for (i=0;i<nodes;i++)
    if (i != curr_node) SyncLink(curr_node, scln_a, i);
  LIST_t& my_adj = ADJLIST(curr_node,curr_node);
  for (i=0;i<my_adj.Length();i++) {
    my_adj.GetElm(i,n,etime);
    SyncLink(curr_node,curr_node,n);
  }//for i
}
```

The Update_Links routine is called when the trigger input from the SatLab update is received. It then scans the new distance matrix and determines if any of our current neighbors have gone out of range. It then finds the x closest satellites, where x is equal to the simulation parameter Maxlinks. If any of these closest satellites also has a free slot, a link is formed between the two satellites. To alert the rest of the routing algorithm that this is a new link, the ordinal number of the new neighbor is increased modulo the number of satellites in the network. (That is, *nodes*+1 is added to the value.)

```
//Function to update the link connections after a Satlab update.
//Does all nodes at once, i,dtable,htable, and neighbors are undefined
//at this point.
//Modifies Neighbors: Leaves entry intact if link is still up, changes
// to -1 if link has gone down, adds new links to free channnels if
// available.  New nodes are flagged by being offset by nodes+1. (See
// below)
void Darting::update_links()
{
//  NeighMatrix = Neighbors;
  int i,j,k; //loop counters
  int x,y; //scratch variables
  float a;

  for (i=0;i<nodes; i++) {
    //Deactivate any links who have gone out of range
    for (j=0;j<MAXLINKS;j++) {
      if  (NeighMatrix[i][j] != -1) {
        a = SATDIST(i,(NeighMatrix[i][j]));
        if (a >= MAXDIST) {
          x=NeighMatrix[i][j];

          // Find our link to Neighbor x and remove it
          y=FindNode(x,ADJLIST(i,i));

          //Find our record of Neighbor x's link to us and remove it
          if (y != -1) delete(ADJLIST(i,i).Remove(y));
          y=FindNode(i,ADJLIST(i,x));
          if (y != -1) delete(ADJLIST(i,x).Remove(y));

          NeighMatrix[i][j] = -1;          //Flag the link as down
        }//if a>= MAXDIST
      }//if N!=-1
    }//for j
  }//for i

  for (i=0;i<nodes; i++) {
    //Recover this node's working environment
    for (j=0;j<MAXLINKS;j++) neighbor[j] = NeighMatrix[i][j];

    //Find closest neighbors, incuding current neighbors
    int best[nodes],tempi;
    float bdist[nodes],tempf;
    float mind = MAXDIST;
    for (j=0;j<nodes;j++) {best[j]=-1; bdist[j]=MAXDIST;}
    for (j=0;j<nodes;j++) {
      a = SATDIST(i,j);
      if (a<mind && a>1.0) {
      x=j;
      for (k=0;k<nodes;k++)
```

```
      if (a<bdist[k]) {
        tempf = bdist[k]; bdist[k] = a; a = tempf;
        tempi = best[k];  best[k] = x;  x = tempi;
        } //if
          mind = a;
    }//if
      }//for j
    //best[] and bdist[] should now have closest nodes in order

    //Delete any candidates that are already neighbors and re-pack list
    for (j=0;j<nodes;j++)
      for (k=0;k<MAXLINKS;k++)
    if (best[j] == neighbor[k] || best[j]+nodes+1 == neighbor[k])
      best[j] = -1;
    x=0; y=1;
    do {
      if (best[x] != -1) {x++; y++;}
      else {
    if (y < nodes && best[y] != -1) {
          tempf = bdist[x]; bdist[x] = bdist[y]; bdist[y] = tempf;
      tempi = best[x];  best[x] = best[y];  best[y] = tempi;
          x++; y++;
     } else {
      y++;
     }//if best[y] != -1
       }//if best[x] != -1
     } while (x < nodes && y < nodes);

    //Try to fill unused links
    int flag = 0;
    x = 0;
    if (best[x] != -1) for (j=0;j<MAXLINKS;j++) {
      if (neighbor[j] == -1) {
    flag = 0;
        while (x < nodes && best[x] != -1 && flag == 0) {
      for (y=0;y<MAXLINKS;y++) { //see if candidate has open link
            if (NeighMatrix[best[x]][y] == -1) {
           //add nodes+1 to flag this as a new entry
          neighbor[j] = best[x]+nodes+1;
          NeighMatrix[best[x]][y] = i+nodes+1;
          flag = 1;
              y=MAXLINKS; //stop the for loop
            }// if free slot
      }// for y in candidate's links
      x++;
    }//while
      }// if n[j]=-1
    }//for j

    //Put back changes
    for (j=0;j<MAXLINKS;j++) NeighMatrix[i][j] = neighbor[j];

}//for i
}//function update_links
```

One of the assumptions that the Darting protocol makes is that all satellites start operation with a knowledge of the topology of the network. To accommodate that, the following function, called during the first network update, uses global network data to determine the initial adjacency lists.

```
//Function Build_Initial_AdjLists
```

```
//Darting assumes all nodes know the initial topology of all links
//when the network is started.
void Darting::Build_Initial_AdjLists()
{
  int i,j,k;
  int neigh;
  LIST_t::QueueOrderings FIFO = (LIST_t::QueueOrderings)0;
  for (i=0;i<nodes;i++) {
    for (j=0;j<nodes;j++) {
      for (k=0;k<MAXLINKS;k++) {
    //At start, all entries from Update_Links should be flagged as new
    neigh = NeighMatrix[j][k]-nodes-1;
    if (neigh > -1) {
      tempint = (INTEGER_t)(SATDIST(j,neigh));
      ADJLIST(i,j).Enqueue(tempint,FIFO,neigh);
      }//if
    }//for k
      }//for j
    }//for i
}//Build initial lists
```

The following functions perform the processing necessary to determine the least cost paths to each network node. The actual implementation here is a form of Dijkstra adapted from [CoL90].

```
//Function Relax - routine from CH25 of the Algorithms book.
inline void Darting::Relax(int n_curr, int n_from, int n_to)
{
  int u,v,w; //variables from text
  u = n_from;
  v = n_to;
  w = (int&)ADJLIST(n_curr,u)[FindNode(v,ADJLIST(n_curr,u))];
  if (RDist[n_curr][v] > RDist[n_curr][u] + w) {
    RDist[n_curr][v] = RDist[n_curr][u] + w;
    Pred[n_curr][v] = u;
  }
}//Relax

//Function to do routing updates
//Implements the Dijkstra routine from [CSCE586????]
void Darting::rt_update(int curr_node)
{
  LIST_t Q;
  int i,j,listlength,u;
  unsigned int v;
  double t;
  LIST_t::QueueOrderings fifo = (LIST_t::QueueOrderings)0;

  //Initialize-Single-Source(G,s)
  for (i=0;i<nodes;i++) {
    RDist[curr_node][i] = (int)MAXDIST;
    Pred[curr_node][i] = -1;
   }//for i
  RDist[curr_node][curr_node] = 0;

  //Q<-V[G]
  for (i=0;i<nodes;i++) {tempint = i;
    Q.Enqueue(tempint,fifo,RDist[curr_node][i]);}

  while (Q.Length()>0) {

    //u<-Extract-Min(Q)
    tempint=(INTEGER_t&)Q[Q.Length()-1]; u=tempint;
```

```
      delete(Q.Remove(Q.Length()-1));

    //for each vertex v E Adj[u]
    listlength = ADJLIST(curr_node,u).Length();
    for(i=0;i<listlength;i++){
    // Get v  (remember, node number is encoded as the priority)
      ADJLIST(curr_node,u).GetElm(i,v,t);
      Relax(curr_node,u,v);

      //Update node v's distance entry in Q
      for (j=0;j<Q.Length();j++) {if ((INTEGER_t&)Q[j]==v) break;}
      if (j < Q.Length()) {
           delete(Q.Remove(j));
           tempint = v;Q.Enqueue(tempint,fifo,RDist[curr_node][v]);
    }//if j<length

      }//for i : v E Adj[u]

    }//while
}//function rt_update
```

The following functions implement the ping packet extension to Darting that was necessary for correct operation of the protocol under our traffic generation distribution. Details of this problem, and its solution can be found in Appendix C.

```
//This routine attempts to determine if we're in a situation where the
//incoming type 3 packet does not contain enough information to resolve
//the problem. To do that, it checks to see if our new calculated cost
//to the destination is the same as the cost passed to us in the update
//packet.  If it is different then we've encountered a degenerate case
//where we've lost contact with what used to be an active path.  To
//solve that, we'll generate a ping packet to retrieve information from
//as far down the defunct path as we can get.
void Darting::Check_Degenerate_Type3(int n_curr, int dest, int n_via, int p_cost)
{
int mypath[nodes], bad_node, pktlen;
int x=0, y=0, k=0; //scratch variables
LIST_t::QueueOrderings FIFO = (LIST_t::QueueOrderings)0;

  //Determine "optimal" path from local data
  int next_hop = -1;
  int Cmin = 0;
  int Cant = 0;
  int j = dest;
  while (j != n_curr) {
    Cant = Cmin;
    mypath[x++] = next_hop = j;
    j = Pred[n_curr][j];
    if (j != -1) {y = FindNode(next_hop, ADJLIST(n_curr,j));}
    else         {y = -1;}
    if (y == -1) {
      j=n_curr;
      next_hop = -1;
    }else{
      Cmin += (INTEGER_t&)ADJLIST(n_curr,j)[y];
    }//if y=-1
   }//while
  x--; //back index up

  if (next_hop == -1) return;  //No path to destination,
                               //can't do anything else.
  if (Cant == p_cost) return;  //Costs match, we should be OK now.

  //If we get here, costs don't match, so we have a problem.
```

```
    //Need to see where our "optimal" path differs from the path
    //sent to us in the update packet, and ping the nodes in
    //discrepancy.
    int p = x;
    VECTOR_t& payload = (*TempPacket)->scl_list;
    while (p > 0) {
      LIST_t& scl_list = (LIST_t&)payload[mypath[p]];
      if (scl_list.Length() == 0) { //We've got a winner.
        bad_node = mypath[p];
        SatDSwdpayload_t *OutPacket;
        OutPacket = new (SatDSwdpayload_t);
        pktlen = (*OutPacket)->packetlength = -PACKETHEADERSIZE;
      //Negative pktlen signals new pkt.

        (*OutPacket)->Cost = Cmin;
        (*OutPacket)->PacketType = 5;  //Ping packet
        (*OutPacket)->source = (*OutPacket)->from_node = n_curr;
        (*OutPacket)->destination = bad_node;
        (*OutPacket)->to_node = n_via;

        LIST_t EmptyList;
        VECTOR_t& outload = (*OutPacket)->scl_list;
        outload.ChangeLengthVECTORDefaultValue(nodes, EmptyList);
        (*OutPacket)->sequencenumber = (*TempPacket)->sequencenumber;

       //Put our SCL data in
        LIST_t& my_scl = (LIST_t&)payload[n_curr];
        x = ADJLIST(n_curr,n_curr).Length();
        if (x != 0) {
          pktlen += x*32;
          unsigned int node=0;
          double time_entered;
          INTEGER_t cost;
          for (k=0;k<x;k++) {
            cost = (INTEGER_t&)ADJLIST(n_curr,n_curr).GetElm(k,node,time_entered);
            my_scl.Enqueue(cost,FIFO,node);
          }//for k
        }//if x

        OutList.Enqueue(*OutPacket);
        delete(OutPacket);
        NextPacket_Entry().Schedule(0,trig);
        //break; //Exit while loop
      }//if scl_length==0
      p--;
    }//while p
  }//Check_Degen

  //This function turns the ping packet around when it has reached
  //it's destination and returns it to the sender.  The packet type
  //is negated to distinguish it from a new ping.
  void Darting::Return_Ping()
  {
    int x,j,k;
    int source = (*TempPacket)->source;
    int n_from = (*TempPacket)->from_node;
    int n_curr = (*TempPacket)->to_node;
    int pktlen = (*TempPacket)->packetlength;
    VECTOR_t& payload = (*TempPacket)->scl_list;
    LIST_t::QueueOrderings FIFO = (LIST_t::QueueOrderings)0;

    (*TempPacket)->source = (*TempPacket)->from_node = n_curr;
    (*TempPacket)->destination = source;
    (*TempPacket)->to_node = n_from;
    (*TempPacket)->PacketType = -5;

        //Put our SCL data in
        LIST_t& my_scl = (LIST_t&)payload[n_curr];
        x = my_scl.Length();
```

```
      if (x > 0) {
        pktlen -= x*32;
        for (j=0;j<x;j++) delete(my_scl.Dequeue());
      }//if
      x = ADJLIST(n_curr,n_curr).Length();
      if (x != 0) {
        pktlen += x*32;
        unsigned int node=0;
        double time_entered;
        INTEGER_t cost;
        for (k=0;k<x;k++) {
          cost = (INTEGER_t&)ADJLIST(n_curr,n_curr).GetElm(k,node,time_entered);
          my_scl.Enqueue(cost,FIFO,node);
        }//for k
      }//if x

    OutList.Enqueue(*TempPacket);
    delete(TempPacket);
    NextPacket_Entry().Schedule(0,trig);
  }//Return_Ping

  //***************************************************************************
  // Run Functions:
  //  The interface for these functions is generated automatically by
  //  Designer when the primitive is created.  The user then fills in the
  //  functionality.
```

The SyncUpdate_Run function handles all the necessary processing when a trigger is received by the router on the port that indicated that the satellites have moved. After calling the *update_links* function, and doing some additional processing on the first iteration, it updates the nodes' adjacency tables and returns.

```
inline void Darting::SyncUpdate_Run(const TRIGGER_t& SyncUpdate)
{
  LIST_t::QueueOrderings FIFO = (LIST_t::QueueOrderings)0;

  gstations = NumberofGroundStations;
  nodes = NumberofSatellites;
  MAXLINKS = MaxLinks;

  int i,j; // loop counters
  int y;   // scratch variables
  simtime = TNow(); seqnum = 1;

  CheckInit();  //See if we need to set up the matricies

  //Process updates for all nodes
  update_links();
  if (simtime==0) Build_Initial_AdjLists();

  //Update each node's adjacency lists
  for (i=0;i<nodes;i++) {

    //Recover this node's working environment
    for (j=0;j<MAXLINKS;j++) neighbor[j] = NeighMatrix[i][j];

    //Process link updates-----------------------------------------------

    for (j=0; j<MAXLINKS; j++) {
    if (neighbor[j] == -1) {;  //Link down
      //update_links should take care of managing the adjacency lists
      }//if lost link
```

```
    if (neighbor[j] < nodes+1 && neighbor[j] > -1) { //Link cost change
      tempint = (INTEGER_t)(SATDIST(i,neighbor[j]));
      y=FindNode(neighbor[j],ADJLIST(i,i));
      ADJLIST(i,i)[y]=tempint;
      y=FindNode(i,ADJLIST(i,neighbor[j]));
      ADJLIST(i,neighbor[j])[y] = tempint;
      }//if old link

    if (neighbor[j] > nodes) { //New neighbor
      neighbor[j] = neighbor[j]-nodes-1; // un-flag node
    //For first iteration, new links are added in
    //Build_Initial_AdjLists
      if (simtime > 0) {
        tempint = (INTEGER_t)(SATDIST(i,neighbor[j]));
        ADJLIST(i,i).Enqueue(tempint,FIFO,neighbor[j]);
        ADJLIST(i,neighbor[j]).Enqueue(tempint,FIFO,i);
    }//if simtime > 0
      }//if new link

    }//for j in MAXLINKS //-----------------------------------------

    rt_update(i);  //Update the routing tables for this node

    //Re-pack node's environment back into the global variables
    for (j=0;j<MAXLINKS;j++) NeighMatrix[i][j] = neighbor[j];

  } //for all nodes (i)

//Output old-style Satcom matrix
int nexthop;
for (i=0;i<nodes;i++) {
  Pred[i][i] = i;  //Make next hop to ourself be ourself
  for (j=0;j<nodes;j++) {
    nexthop = Pred[i][j];
    if (nexthop != -1) while (Pred[i][nexthop] != i) nexthop = Pred[i][nexthop];
    if (nexthop == i) nexthop = j; //Correct for immediate neighbors
    RouteMatrix[i*nodes+j] = nexthop;
    }//for j
  }//for i
SetRoutingtablememory(RouteMatrix);

}
//*********************************************************************
```

Inbound_Update_Run is called every time a packet is passed to the routing algorithm during normal network operations. It checks the packet type and forwards it to its destination if it is a data packet. Update packets are handled as described in [TsM95] with B=1. Ping packets are handled as described in Appendix C.

```
inline void Darting::inbound_update_Run(const SatDSwdpayload_t& inbound_update)
{

  RouteMatrix = Routingtablememory;
  gstations = NumberofGroundStations;
  nodes = NumberofSatellites;
  MAXLINKS = MaxLinks;
  LIST_t::QueueOrderings FIFO = (LIST_t::QueueOrderings)0;

  TempPacket = (SatDSwdpayload_t*)inbound_update.CopyArc();

  int n_from, n_curr, p_cost, pkttype, pktlen, update_flag;
```

```
int i,j,k,x,y; // loop counters, etc
simtime = TNow();

int g = (*TempPacket)->sequencenumber;

int d_dstn = (*TempPacket)->destination;
int s_srce = (*TempPacket)->source;
pkttype = (*TempPacket)->PacketType;
pktlen = (*TempPacket)->packetlength;

//If noone has reset the packetlength, we need to take care of it.
if (pktlen < 0) {pktlen = -pktlen; (*TempPacket)->packetlength = pktlen;}

//Kludge for Ping packets, we'll put it back later
if (pkttype == 5 || pkttype == -5) pkttype = 2;

if (pkttype > 3 || d_dstn >= nodes || (*TempPacket)->source >= nodes)
 { //Drop any unknown type or type 4 packets that make it back here
  delete(TempPacket);
  return;
 }

VECTOR_t& payload = (*TempPacket)->scl_list;

//Fill in routing overhead fields for new packets
if (pkttype == 1) {
  (*TempPacket)->Cost = 0;
  LIST_t EmptyList;
  payload.ChangeLengthVECTORDefaultValue(nodes, EmptyList);
  (*TempPacket)->from_node = (*TempPacket)->source;
  (*TempPacket)->to_node   = (*TempPacket)->source;
  (*TempPacket)->sequencenumber = seqnum++;
  }

n_from = (*TempPacket)->from_node;
n_curr = (*TempPacket)->to_node;
p_cost = (*TempPacket)->Cost;
if ((*TempPacket)->source == (*TempPacket)->destination) {
  //Sending data to ourself, drop the packet
  (*TempPacket)->PacketType = 4;
  OutList.Enqueue(*TempPacket);
  delete(TempPacket);
  NextPacket_Entry().Schedule(0,trig);
  return;
  }

//Step 0 - Go through steps 3-5 with i=0 if this is a new message

//Step 1 - Incorperate p_status into local cost matix
update_flag = 0;
if (pkttype != 1) {
  for (j=0;j<nodes;j++) {
    LIST_t& scl_list = (LIST_t&)payload[j];
    if (j != n_curr && scl_list.Length() > 0) {
     if (scl_list == ADJLIST(n_curr,j)) {
     //do nothing
      }else{
     update_flag = 1;
     x = ADJLIST(n_curr,j).Length();
       for (k=0;k<x;k++) delete (ADJLIST(n_curr,j).Dequeue());
     x = scl_list.Length();
       unsigned int node=0;
       double time_entered;
       INTEGER_t cost;
       for (k=0;k<x;k++) {
        cost = (INTEGER_t&)scl_list.GetElm(k,node,time_entered);
      ADJLIST(n_curr,j).Enqueue(cost,FIFO,node);
        }//for k
       SyncLinks(n_curr,j);
```

```
    }//if scl_list==ADJLIST
     }//if scl_list.Length > 0
    }//for j
  if (update_flag == 1) rt_update(n_curr);
  }// if not new packet

 if (pkttype == 3) Check_Degenerate_Type3(n_curr,s_srce,n_from,p_cost);

 //Handle Ping packets
 pkttype = (*TempPacket)->PacketType; //Restore correct value
 if (pkttype == 5 && (*TempPacket)->to_node == (*TempPacket)->destination) {
   Return_Ping();
   return;
 }//ping packet

 //Step 2 - If this is the destination, give data to user and stop
 if ((*TempPacket)->destination == n_curr) {
   (*TempPacket)->PacketType = 4;
   OutList.Enqueue(*TempPacket);
   delete(TempPacket);
   NextPacket_Entry().Schedule(0,trig);
   return;
   }

 //Recover this node's working environment
 for (j=0;j<MAXLINKS;j++) neighbor[j] = NeighMatrix[n_curr][j];
 int link =-1;
 for (k=0;k<MAXLINKS;k++) { //What link is source on?
   if (neighbor[k] == n_from) link=k;
   }//for k

 //Step 3 - Determine optimal path
int next_hop = -1;
int Cmin = 0;
int Cant = 0;
if (pkttype != 3) {
 j = d_dstn;
 while (j != n_curr) {
   Cant = Cmin;
   next_hop = j;
   j = Pred[n_curr][j];
   if (j != -1) {y = FindNode(next_hop, ADJLIST(n_curr,j));}
   else         {y = -1;}
   if (y == -1) {
     j=n_curr;
     next_hop = -1;
   }else{
     Cmin += (INTEGER_t&)ADJLIST(n_curr,j)[y];
   }//if y=-1
  }//while
 }//if not control packet

//Step 4 - Assemble outgong data message
if (pkttype != 3 && next_hop != -1) {
  (*TempPacket)->from_node = n_curr;
  (*TempPacket)->to_node = next_hop;
  (*TempPacket)->Cost = Cant;
  if (pkttype == 1) (*TempPacket)->PacketType = 2;
  LIST_t& scl_list = (LIST_t&)payload[n_curr];
  x = scl_list.Length();
  if (x > 0) {
  pktlen -= x*32;
    for (j=0;j<x;j++) delete(scl_list.Dequeue());
    }//if
  x = ADJLIST(n_curr,n_curr).Length();
  pktlen += x*32;
  unsigned int node=0;
  double time_entered;
  INTEGER_t cost;
```

```
    for (k=0;k<x;k++) {
      cost = (INTEGER_t&)ADJLIST(n_curr,n_curr).GetElm(k,node,time_entered);
      scl_list.Enqueue(cost,FIFO,node);
      }//for k
    (*TempPacket)->packetlength = pktlen;

    //Step 5 - Transmit packet to next node
    OutList.Enqueue(*TempPacket);
    delete(TempPacket);
    NextPacket_Entry().Schedule(0,trig);

  }else{  //No path to destination, or control packet, drop the packet.
    (*TempPacket)->PacketType = 4;
    OutList.Enqueue(*TempPacket);
    delete(TempPacket);
    NextPacket_Entry().Schedule(0,trig);
    return;
  }//if pkttype !=3 && nexthop != 1

  //Step 6 - Check if predecessor update required
  if (Cmin != p_cost && (pkttype == 2 || pkttype == 5)) {
    TempPacket = (SatDSwdpayload_t*)inbound_update.CopyArc();
    VECTOR_t& payload = (*TempPacket)->scl_list;
    (*TempPacket)->Cost = Cmin;
    (*TempPacket)->PacketType = 3;
    (*TempPacket)->source = d_dstn;
    (*TempPacket)->from_node = n_curr;
    (*TempPacket)->destination = (*TempPacket)->to_node = n_from;
    (*TempPacket)->sequencenumber = -(*TempPacket)->sequencenumber;
    pktlen = PACKETHEADERSIZE;
    for (k=0;k<nodes;k++) {
      LIST_t& scl_list = (LIST_t&)payload[k];
      x = scl_list.Length();
      if (x > 0) for (j=0;j<x;j++) delete(scl_list.Dequeue());
      }//for k
    j=d_dstn;
    while (j != n_curr) {
      LIST_t& scl_list = (LIST_t&)payload[j];
      x = ADJLIST(n_curr,j).Length();
      pktlen += x*32;
      unsigned int node=0;
      double time_entered;
      INTEGER_t cost;
      for (k=0;k<x;k++) {
        cost = (INTEGER_t&)ADJLIST(n_curr,j).GetElm(k,node,time_entered);
        scl_list.Enqueue(cost,FIFO,node);
        }//for k
      j = Pred[n_curr][j];
      if (j == -1) { //if j=-1 then there is no path to the destination
        j = n_curr;  //exit from the loop
        }// if j
    }//while
    //Put our SCL data in
    x = ADJLIST(n_curr,n_curr).Length();
    if (x != 0) {
      pktlen += x*32;
      LIST_t& scl_list = (LIST_t&)payload[n_curr];
      unsigned int node=0;
      double time_entered;
      INTEGER_t cost;
      for (k=0;k<x;k++) {
        cost = (INTEGER_t&)ADJLIST(n_curr,n_curr).GetElm(k,node,time_entered);
        scl_list.Enqueue(cost,FIFO,node);
        }//for k
      }//if x

    //Negative packetlength signals new packet to external handlers
    (*TempPacket)->packetlength = -abs(pktlen);
    OutList.Enqueue(*TempPacket);
```

```
    delete(TempPacket);
    NextPacket_Entry().Schedule(0,trig);
  }//if not initial packet or control packet, and predecessor update is required

  //Re-pack node's environment back into the global variables
  for (j=0;j<MAXLINKS;j++) NeighMatrix[n_curr][j] = neighbor[j];

//Output old-style Satcom matrix
  int nexthop;
  Pred[n_curr][n_curr] = n_curr;  //Make next hop to ourself be ourself
  for (j=0;j<nodes;j++) {
    nexthop = Pred[n_curr][j];
    if (nexthop != -1) while (Pred[n_curr][nexthop] != n_curr) nexthop =
Pred[n_curr][nexthop];
    //Correct for immediate neighbors
    if (nexthop == n_curr) nexthop = j;
    RouteMatrix[n_curr*nodes+j] = nexthop;
    }//for j
  SetRoutingtablememory(RouteMatrix);

}
//*********************************************************************
// Asynchronous Functions:
```

This NextPacket_Run outputs any packets in the output queue. When each packet is enqueued, a trigger is placed on the event list that activates this function.

```
inline void Darting::NextPacket_Run(const TRIGGER_t& NextPacket)
{
  SatDSwdpayload_t *OutPacket;
  int i = OutList.Length();
  if (i>0) {
    OutPacket = (SatDSwdpayload_t *)OutList.Dequeue();
    outbound_update(*OutPacket);
    delete(OutPacket);
    }//if i>1
}

/****  User Code Above Here ****/
```

B.2 Extended Bellman Ford

Much of the interfacing code for Bellman Ford is identical to the Darting code, with the exception that Bellman Ford does not need access to the actual data packets in the network. It therefore has no conception of packet types. Any packet it receives is considered to be an update packet, and any packet it transmits is likewise an update packet.

The algorithm itself is adapted from Cheng, et. al [ChR89]. The portions of the code that are identical to Darting are presented with minimal, if any, additional description.

o External Ports:

Input:

SyncUpdate is of type TRIGGER: Description: Triggers bulk link cost updates using the latest data from SatLab. The primitive will iterate through every node and generate any necessary updates.

inbound_update is of type Sat DS w/ payload: Description: Accepts input Sat DS w/ payload in which the payload is an asynchronous input vector used to accept routing update messages from the network. Each "element" of the vector is a triple, (j, D_{kj}, H_{kj}). Therefore, the first three numbers in the vector correspond to the first "element", numbers 4-6 correspond to the second element, etc. The C++ code expects the numbers to be formatted in this manner.

Output:

outbound_update is of type Sat DS w/ payload: Description: Outputs Sat DS w/ payload packets with appropriate topology update messages based upon the input changes. One input may result in many output packets.

routematrix is of type INT-VECTOR: Description: Outputs a Satcom_dbs style global route table based upon any input changes. Seeing whereas the primitive will update the global memory anyway, though, this output is fairly useless, and is only included for backward compatibility with Designer.

o External Arguments:

(M) "Routing table memory" is of type "INT-VECTOR"

(M) "Elevation Table Memory" is of type "REAL-VECTOR"

(M) "Distance table memory" is of type "REAL-VECTOR"

(M) "Number of Satellites" is of type "INTEGER"

(M) "Number of Mobiles" is of type "INTEGER"

(M) "Mobile Latitude Table Memory" is of type "REAL-VECTOR"

(M) "Mobile Altitude Table Memory" is of type "REAL-VECTOR"

(M) "Number of GroundStations" is of type "INTEGER"

(P) "MaxLinks" is of type "INTEGER"

o Internal Arguments:

(M) "RouteDistance" is of type "REAL-VECTOR"

Description: Holds each node's estimated distance to each destination.

Initialization Value: Vector length: (1) Initial Value: 0.0

(M) "NodeMemory" is of type "VECTOR"

Description: Stores routing state from iteration to iteration.

Initialization Value: Vector length: (1) Initial Value: Uninitialized

(M) "OutputQueue" is of type "VECTOR"

Description: Holds packets awaiting output from each node.

Initialization Value: Vector length: (1) Initial Value: Uninitialized

(E) "NextPacket" is of type "EVENT-LIST"

Initialization Value: Uninitialized

Description: Used to schedule a simulation event to output packets from OutputQueue.

(M) "Neighbors" is of type "INT-VECTOR"

Description: Holds the state of the network links.

Initialization Value: Vector length: (1) Initial Value: 0

```
/*
 *        Module Name :                exBF
 *        Template Created By :        3.0
 *        Author:                      rjanoso
 *        Last Modification Date:      30-Jul-1996 17:16:36
 *        Template Date:               30-Jul-1996 17:16:46
 */

/****  Includes and Defines Below Here ****/
#include <LIST.hh>
#define  SATDIST(X,Y) Distancetablememory[gstations*(gstations+nodes) +
X*(gstations+nodes)+(gstations+Y)]
#define PACKETHEADERSIZE 256
/****  Includes and Defines Above Here ****/
```

The data structures used by Cheng have been divided into two matrices: *dtable*, which holds the distances to each node in the network through each of the outgoing links, and *htable*, which holds the header (predecessor) node for each destination along the path out the respective link. The *routes* matrix holds the optimal paths determined by the algorithm for each node. Each entry is a triple, consisting of outgoing link, total distance, and header(predecessor) node along the path.

```
/****  Instance Definitions Below Here ****/
  void Init();
  int  in_path(int n_from, int l_via, int n_to);
  int  dtable_min(int row);
  void rt_update(int curr_node);
  void route_all();
  void update_links();

  INTVECTOR_t tempiv;
  TRIGGER_t trig;

  VECTOR_t nodemem;
```

```
  LIST_t OutList;
  SatDSwdpayload_t *TempPacket;
  INTVECTOR_t RouteMatrix, NeighMatrix, Vi, payload;
  REALVECTOR_t RDist;

  double simtime, iter_time;
  int gstations,seqnum;
  int nodes;
  int MAXLINKS;
  int   **htable;
  int   **routes;
  float **dtable;
  int    *neighbor;
  const float MAXDIST = 1000000000.0;

  int initflag;
/****  Instance Definitions Above Here ****/


/****  User Constructor Code Below Here ****/
initflag = -1;
iter_time = 1000000;
DEBUGTIME = 1000000;
/****  User Constructor Code Above Here ****/


/****  User Code Below Here ****/

void exBF::Init()
{
  EVENTLIST_t Temp = NextPacket;
  Temp.Extend(NextPacket_Entry());
  SetNextPacket(Temp);
}
```

The in_path function is used by Cheng's algorithm to determine if the optimal path to a specified node goes out the indicated link.

```
//Function to determine if my path to n_to goes out through l_via
// n_'s are node numbers, l_'s are indexed into the neighbor/htable columns.
int exBF::in_path(int n_from, int l_via, int n_to)
{
  int h;
  if (n_to > nodes || n_to < 0 || l_via < 0 || l_via > MAXLINKS) return 0;
  //routes[][2] is the head node along our best path to n_to.
  h = routes[n_to][2];
  if (h < 0 || h > nodes)  return 0;
  if (h == n_from)         return 0;
  if (h == neighbor[l_via])        return 1;
  else                             return in_path(n_from,l_via,h);
}//function in_path
```

Dtable_min is used by the routing algorithm when determining what outgoing link has the shortest destination to the desired destination.

```
//Function to determine the minimum entry in a dtable row
//returns an integer index to the minimum entry
int exBF::dtable_min(int row)
{
  int x; //Loop Counters
```

```
  int minnode = -1;
  float mindist = MAXDIST;
  for (x=0;x<MAXLINKS;x++) {
    if (dtable[row][x] < (int)mindist) {
    minnode = x;
    mindist = dtable[row][x];
    }//if
  }//for
  return(minnode);
}//function dtable_min
```

Rt_update performs the actual routing function. Operation is as described by Cheng in [ChR89]. This basically consists of checking each row in *dtable* and recording the link with the smallest entry as the preferred output link for packets to that destination. The notable exception to this is that no source node will advertise a path to a neighbor when that neighbor lies along the path to the destination node. This prevents several nasty looping effects present in the original Bellman-Ford algorithm.

```
//Function to do routing updates
// Selects perferred neighbors from minimum of dtable entries for that
// node's row as long as each node along the path to the desination has
// it's shortest path also passing through the candidate neighbor.
//  -- Destroys routes, outvect
void exBF::rt_update(int curr_node)
{
  //Initialize variables:
  const int UNMARKED = -1;
  const int UNDETERMINED = MAXLINKS+2;
  float mind;
  int flag1=0,flag2=0,flag3=0;
  int b,c,p,x,mincol; //Loop Counters, misc indicies
  int path[nodes+1]; //holds reconstructed paths from htable
  for (x=0;x<nodes;x++)    {routes[x][0] = UNMARKED;
                            routes[x][1] = (int)MAXDIST;
                            routes[x][2] = UNMARKED;}
  //Fix up the entry to ourself
  routes[curr_node][0] = MAXLINKS+1; //No outgoing link to ourself
  routes[curr_node][1] = 0;                //Zero cost to transmit
  routes[curr_node][2] = UNMARKED;   //path to ourself has no header

  //Re-calculate routing data for each node:
  for (x=0;x<nodes;x++) {

    b = dtable_min(x);
    if (b == -1 || dtable[x][b] == (int)MAXDIST) {
      //Noone we know of has a path to that node node
      if (x != curr_node) routes[x][0] = UNDETERMINED;

     } else {

      //Reconstruct path from htable entries
      p = -1; c = x; flag1 = flag2 = flag3 = 0;

      if (c != curr_node) do {
       c = htable[c][b];
       if (c != -1) {
      path[++p] = c;

      mincol = dtable_min(c); //Returns -1 if no path
```

```
      if (mincol == -1) {mind = MAXDIST;}
        else {mind = dtable[c][mincol];}

      flag1 = (dtable[c][b] > mind && c != curr_node);
      flag2 = (htable[c][b] == curr_node);
      flag3 = (routes[c][0] != UNMARKED);

       }else{

        flag1 = (1 > 0); //Set flag1 true
       }//if c != -1

      } while  (!flag1 && !flag2 && !flag3);

      if (flag1 || routes[c][0] == UNDETERMINED) {
        routes[x][0] = UNDETERMINED;
        routes[x][1] = (int)MAXDIST;
      } else {
      routes[x][0] = b; //Preferred Neighbor
      routes[x][1] = (int)dtable[x][b]; //Distance to x via b
      routes[x][2] = htable[x][b]; //Head of path to x via b
       }//if

     }//if path exists

  }//for x (all nodes)

  //Store the updated tables in the output vector
  Vi.ChangeLength(nodes*3,-1);
  for (x=0;x<nodes;x++) {
    Vi[x*3+0] = x; //node #
    Vi[x*3+1] = routes[x][1]; //distance to node
    Vi[x*3+2] = routes[x][2]; //head node of path
    }//for x

}//function rt_update
```

Due to the huge number of update packets that exBF generated when required to converge to a global topology from a completely blank slate at network startup, route_all was added to cheat and synchronize all nodes with global information during the first iteration. This modification saves several hours of simulation time, and as the first 60 seconds of simulation data are thrown out anyway, does not affect the comparison at all.

```
//Function to synchronize all routing tables to global info
// Uses the Dijkstra algorithm from Darting.
void exBF::route_all()
{
  LIST_t Q;
  LIST_t::QueueOrderings FIFO = (LIST_t::QueueOrderings)0;
  INTEGER_t int_t;
  int i,j,u,v,w,curr_node;
  int **Pred, **RDist;

  nodemem = NodeMemory;
  NeighMatrix = Neighbors;

  Pred = new int*[nodes];
  if (Pred == 0) {cerr << "Out of Memory!";}
  for (i=0;i<nodes;i++) {
    Pred[i] = new int[nodes];
```

```
  if (Pred[i] == 0) {cerr << "Out of Memory!";}
  }//for i

RDist = new int*[nodes];
if (RDist == 0) {cerr << "Out of Memory!";}
for (i=0;i<nodes;i++) {
  RDist[i] = new int[nodes];
  if (RDist[i] == 0) {cerr << "Out of Memory!";}
  }//for i

//First build global tables
for(curr_node=0;curr_node<nodes;curr_node++) {

  //Init Single Source Graph(G,s)
  for (i=0;i<nodes;i++) {
    RDist[curr_node][i] = (int)MAXDIST;
    Pred[curr_node][i] = -1;
    }//for i
  RDist[curr_node][curr_node] = 0;
  Pred[curr_node][curr_node] = curr_node;

  //Q<-v[G]
  for (i=0;i<nodes;i++) {
  int_t = i;
    Q.Enqueue(int_t,FIFO,RDist[curr_node][i]);}

  while (Q.Length()>0) {

    //u<-Extract-Min(Q)
    int_t=(INTEGER_t&)Q[Q.Length()-1]; u=int_t;
    delete(Q.Remove(Q.Length()-1));

    //for each vertex v E Adj[u]
    for (i=0;i<MAXLINKS;i++) {
      v = NeighMatrix[u*MAXLINKS+i];
      if (v > nodes) {v = v-nodes-1; NeighMatrix[u*MAXLINKS+i] = v;}
      if (v != -1) {
    //Relax(u,v,w);
        w = (int)SATDIST(u,v);
        if (RDist[curr_node][v] > RDist[curr_node][u] + w) {
          RDist[curr_node][v] = RDist[curr_node][u] + w;
          Pred[curr_node][v] = u;
      }
        //Update node v's distance entry in Q
        for (j=0;j<Q.Length();j++) {if ((INTEGER_t&)Q[j]==v) break;}
        if (j < Q.Length()) {
          delete(Q.Remove(j));
              int_t = v;Q.Enqueue(int_t,FIFO,RDist[curr_node][v]);
      }//if j<length
        }//if v != -1
      }//for each vertex v E Adj[u]

    }//while

  }//for curr_node

//Now fill in each node's local memory with the global info
for(curr_node=0;curr_node<nodes;curr_node++) {
  INTVECTOR_t& iv = (INTVECTOR_t&)nodemem[curr_node];
  int n,d_cn;

  for (i=0;i<MAXLINKS;i++) {
    n = NeighMatrix[curr_node*MAXLINKS+i];
    if (n > nodes) {
    n = n-nodes-1;
    NeighMatrix[curr_node*MAXLINKS+i] = n;}

    if (n != -1) {
      d_cn = SATDIST(curr_node,n);
```

```
    //Set up tables for in_path
    int link = -1;
    for (j=0;j<MAXLINKS;j++) {
      neighbor[j] = NeighMatrix[n*MAXLINKS+j];
      if (neighbor[j] == curr_node) link = j;
      }// for j
    if (link == -1) {
        printf("Error, no reciprocal link while cheating! curr_node=%i,
                n=%i\n",curr_node,n);
        TerminateSim();}
    for (j=0;j<nodes;j++) routes[j][2] = Pred[n][j];

    //Now fill in dtable and htable.
    //Note: nodemem[j,i] = dtable[j][i]
    //and nodemem[j+nodes,i] = htable[j][i]

    for (j=0;j<nodes;j++) {
      if (in_path(n,link,j)) {
        iv[j*MAXLINKS+i] = (INTEGER_t)((int)MAXDIST);
        iv[nodes*MAXLINKS + j*MAXLINKS+i] = (INTEGER_t)(-1);
        }else{
        iv[j*MAXLINKS+i] = (INTEGER_t)(RDist[n][j] + d_cn);
        if (iv[j*MAXLINKS+i] > (int)MAXDIST)
             iv[j*MAXLINKS+i] = (int)MAXDIST;
        iv[nodes*MAXLINKS + j*MAXLINKS+i] = (INTEGER_t)Pred[n][j];
           }//if in-path
     }//for j = destination node

    //Never go out a link to get to ourself
    iv[curr_node*MAXLINKS+i] = (int)MAXDIST;

    //head node to neighbors is ourself
      iv[nodes*MAXLINKS + n*MAXLINKS+i] = (INTEGER_t)curr_node;

     }//if n!= -1
    }//for i = via link
   }//for curr_node

  SetNodeMemory(nodemem);
  SetNeighbors(NeighMatrix);

  //Delete the dynamically allocated memory
  for (i=0;i<nodes;i++) {
    delete[] RDist[i];
    delete[] Pred[i];
   }//
}//route_all


//Function to update the link connections after a Satlab update.
// Does all nodes at once, i,dtable,htable, and neighbors are undefined
// at this point.  Modifies Neighbors: Leaves entry intact if link is
// still up, changes to -1 if link has gone down, adds new links to free
// channnels if available.  New nodes are flagged by being offset by
// nodes+1.
void exBF::update_links()
{
  NeighMatrix = Neighbors;
  int i,j,k; //loop counters
  int x,y; //scratch variables
  float a;

  for (i=0;i<nodes; i++) {
    //Deactivate any links who have gone out of range
    for (j=0;j<MAXLINKS;j++) {
      if  (NeighMatrix[i*MAXLINKS+j] != -1) {
        a = SATDIST(i,NeighMatrix[i*MAXLINKS+j]);
        if (a >= MAXDIST) NeighMatrix[i*MAXLINKS+j] = -1;
```

```
    }//if
   }//for j
 }//for i

for (i=0;i<nodes; i++) {
  //Recover this node's working environment
  for (j=0;j<MAXLINKS;j++) neighbor[j] = NeighMatrix[i*MAXLINKS+j];

  //Find closest maxlinks neighbors, incuding current neighbors
  int best[nodes],tempi;
  float bdist[nodes],tempf;
  float mind = MAXDIST;
  for (j=0;j<nodes;j++) {best[j]=-1; bdist[j]=MAXDIST;}
  for (j=0;j<nodes;j++) {
    a = SATDIST(i,j);
    if (a<mind && a>1.0) {
  x=j;
  for (k=0;k<nodes;k++)
    if (a<bdist[k]) {
      tempf = bdist[k]; bdist[k] = a; a = tempf;
      tempi = best[k];  best[k] = x;  x = tempi;
      } //if
        mind = a;
  }//if
    }//for j
  // -- best[] and bdist[] should now have closest nodes

  //Delete any candidates that are already neighbors and re-pack list
  for (j=0;j<nodes;j++)
    for (k=0;k<MAXLINKS;k++)
  if (best[j] == neighbor[k] || best[j]+nodes+1 == neighbor[k])
    best[j] = -1;
  x=0; y=1;

  do {
    if (best[x] != -1) {x++; y++;}
    else {
  if (y < nodes && best[y] != -1) {
        tempf = bdist[x]; bdist[x] = bdist[y]; bdist[y] = tempf;
    tempi = best[x];  best[x] = best[y];  best[y] = tempi;
        x++; y++;
   } else {
    y++;
   }//if best[y] != -1
     }//if best[x] != -1
   } while (x < nodes && y < nodes);

  //Try to fill unused links
  int flag = 0;
  x = 0;
  if (best[x] != -1) for (j=0;j<MAXLINKS;j++) {
    if (neighbor[j] == -1) {
  flag = 0;
      while (x < nodes && best[x] != -1 && flag == 0) {
    for (y=0;y<MAXLINKS;y++) { //see if candidate has open link
          if (NeighMatrix[best[x]*MAXLINKS+y] == -1) {
        neighbor[j] = best[x]+nodes+1;
        NeighMatrix[best[x]*MAXLINKS+y] = i+nodes+1;
        flag = 1;
            y=MAXLINKS; //stop the for loop
          }// if free slot
    }// for y in candidate's links
    x++;
  }//while
    }// if n[j]=-1
  }//for j

  //Put back changes
  for (j=0;j<MAXLINKS;j++) NeighMatrix[i*MAXLINKS+j] = neighbor[j];
```

```
  }//for i
  SetNeighbors(NeighMatrix);

}//function update_links


//***************************************************************************
// Run Functions:
```

SyncUpdate_Run differs slightly from Darting in that it must create update packets to begin the convergence iteration. It cycles through each node, updates its out going links, and informs its neighbors of any relevant changes.

```
inline void exBF::SyncUpdate_Run(const TRIGGER_t& SyncUpdate)
{
  nodemem = NodeMemory;
  RouteMatrix = Routingtablememory;
  NeighMatrix = Neighbors;
  RDist = RouteDistance;

  gstations = NumberofGroundStations;
  nodes = NumberofSatellites;
  MAXLINKS = MaxLinks;

  int i,j,k; // loop counters

  simtime = TNow(); seqnum = 1;

  //Initialize global variables if needed
  if (initflag == -1) {
    htable = new int*[nodes];
    if (htable == 0) {cerr << "Out of Memory!";}
    for (i=0;i<nodes;i++) {
      htable[i] = new int[MAXLINKS];
      if (htable[i] == 0) {cerr << "Out of Memory!";}
      }//for i

    dtable = new float*[nodes];
    if (dtable == 0) {cerr << "Out of Memory!";}
    for (i=0;i<nodes;i++) {
      dtable[i] = new float[MAXLINKS];
      if (dtable[i] == 0) {cerr << "Out of Memory!";}
      }//for i

    routes = new int*[nodes];
    if (routes == 0) {cerr << "Out of Memory!";}
    for (i=0;i<nodes;i++) {
      routes[i] = new int[3];
      if (routes[i] == 0) {cerr << "Out of Memory!";}
      }//for i

    neighbor = new int[MAXLINKS];
    if (neighbor == 0) {cerr << "Out of Memory!";}

    initflag = 0;
    }//if initflag

  if (RDist.Length() == 1) RDist.ChangeLength(nodes*nodes, MAXDIST);
  if (RouteMatrix.Length() == 1) RouteMatrix.ChangeLength(nodes*nodes,-1);
  if (NeighMatrix.Length() == 1) {
    NeighMatrix[0] = -1;
    NeighMatrix.ChangeLength(nodes*MAXLINKS,-1);
    SetNeighbors(NeighMatrix);
```

```
    }
   if (nodemem.Length() == 1) {
     nodemem.ChangeLength(nodes);
     tempiv.ChangeLength(2*nodes*MAXLINKS, (int)MAXDIST);
     tempiv[0] = (int)MAXDIST;
     for (j=nodes*MAXLINKS;j<2*nodes*MAXLINKS;j++) tempiv[j] = -1;
     for (j=0;j<nodes;j++) nodemem[j] = tempiv;
     SetNodeMemory(nodemem);
     }

   //Process updates for all nodes
   update_links();
   if (simtime == 0) {route_all();  nodemem = NodeMemory;}
   NeighMatrix = Neighbors;
   for (i=0;i<nodes;i++) {

     //Recover dtable and htable from NodeMemory
     tempiv = (INTVECTOR_t&)nodemem[i];
     for (j=0;j<nodes;j++)
       for (k=0;k<MAXLINKS;k++)
         dtable[j][k] = tempiv[j*MAXLINKS+k];
     for (j=0;j<nodes;j++)
       for (k=0;k<MAXLINKS;k++)
         htable[j][k] = tempiv[nodes*MAXLINKS + j*MAXLINKS+k];
     for (k=0;k<MAXLINKS;k++)
       htable[i][k] = i;

     //Recover this node's working environment
     for (j=0;j<MAXLINKS;j++) neighbor[j] = NeighMatrix[i*MAXLINKS+j];

//+**************
     //Update Neighbors -- Somewhat like Garcia (4)
     float newd, oldd, deltad;

   if (simtime > 0) { //We're cheating first time through
     for (j=0; j<MAXLINKS; j++) {

     //Check for link down
     if (neighbor[j] == -1) {
       for (k=0;k<nodes;k++) {dtable[k][j] = MAXDIST; htable[k][j] = -1;}
       htable[i][j] = i;
       }//if lost link

     //Check if new cost on existing link
     if (neighbor[j] < nodes+1 && neighbor[j] > -1) {
       newd = SATDIST(i,neighbor[j]);
       oldd = RDist[i*nodes+neighbor[j]];
       deltad = newd - oldd;
       for (k=0;k<nodes;k++)
         if (dtable[k][j] < MAXDIST) dtable[k][j]=dtable[k][j] + deltad;
       dtable[neighbor[j]][j] = newd; //Kludge
       }//if old link

     //Check if a new link has been established
     if (neighbor[j] > nodes) {
       neighbor[j] = neighbor[j]-nodes-1; // un-flag node
       newd = SATDIST(i,neighbor[j]);
       for (k=0;k<nodes;k++) {dtable[k][j] = MAXDIST; htable[k][j] = -1;}
       dtable[neighbor[j]][j] = newd;
       htable[neighbor[j]][j] = i;
       htable[i][j] = i; //Head node to ourself is ourself.
       }//if new link

     }//for j in MAXLINKS
   }//if
//+**************

     //Update routing table from dtable -- Garcia (2)
     // This is an unconditional update because this routine handles
```

```
    // Satlab(global) updates, so the k=Pij condition in Garcia's
    // algorithm will always hold.
    rt_update(i);

//Send updates to neighbors -- Garcia (3)
    int b,t;
    INTVECTOR_t outvect; outvect.ChangeLength(nodes*3,-1);

    for (b=0;b<MAXLINKS;b++) {
      if (neighbor[b] != -1) {
       for (t=0;t<nodes;t++) {
        if (in_path(i,b,Vi[t*3])) {
      outvect[t*3+0] = Vi[t*3+0];    //node
      outvect[t*3+1] = (int)MAXDIST; //if in path send infinity
      outvect[t*3+2] = -1;          //and invalid head node
         } else {
      outvect[t*3+0] = Vi[t*3+0];    //node
      outvect[t*3+1] = Vi[t*3+1];    //distance
      outvect[t*3+2] = Vi[t*3+2];    //head node
     }//if in_path
    }//for t -- all nodes
      TempPacket = new (SatDSwdpayload_t);
      (*TempPacket)->source = i;
      (*TempPacket)->sequencenumber = seqnum++;
      (*TempPacket)->Payload = outvect;
      (*TempPacket)->packetlength = outvect.Length()*32 + PACKETHEADERSIZE;
      (*TempPacket)->destination = neighbor[b];
      OutList.Enqueue(*TempPacket);
      delete(TempPacket);
      NextPacket_Entry().Schedule(0,trig);
      }//if neighbor[b] != -1
     }//for b -- all neighbors

    //Re-pack node's environment back into the global variables
    for (j=0;j<nodes;j++)
      for (k=0;k<MAXLINKS;k++)
        tempiv[j*MAXLINKS+k] = (int)dtable[j][k];
    for (j=0;j<nodes;j++)
      for (k=0;k<MAXLINKS;k++)
        tempiv[nodes*MAXLINKS + j*MAXLINKS+k] = htable[j][k];
    nodemem[i] = tempiv;
    SetNodeMemory(nodemem);
    for (j=0;j<MAXLINKS;j++) NeighMatrix[i*MAXLINKS+j] = neighbor[j];
    SetNeighbors(NeighMatrix);

    //Update the Satcom-style RouteMatrix (matrix of next-hops)
    for (j=0;j<nodes;j++) {
     if (i != j) {

    if (routes[j][0] != MAXLINKS+2 && routes[j][0] != -1) {
      //If the destination is flagged, we don't have a path yet, or a
      // link has failed.
      RouteMatrix[i*nodes+j] = neighbor[routes[j][0]];
     } else {
      RouteMatrix[i*nodes+j] = -1;
     }//if destination is not flagged

     } else {

        RouteMatrix[i*nodes+j]=i;

     }//if (i != j)

      RDist[i*nodes+j] = routes[j][1];
    } //for j

    SetRoutingtablememory(RouteMatrix);
    SetRouteDistance(RDist);
```

```
  } //for all nodes (i)

//Output old-style Satcom matrix
routematrix(RouteMatrix);

}//SyncUpdate_Run

//***************************************************************************
```

Inbound_update_run processes incoming updates from our neighbors. The incoming data is merged into our local tables. The algorithm then determines if this new data has made any significant changes, and if so, sends an update out to all of the current node's neighbors.

```
inline void exBF::inbound_update_Run(const SatDSwdpayload_t& inbound_update)
{
  nodemem = NodeMemory;
  RouteMatrix = Routingtablememory;
  NeighMatrix = Neighbors;
  RDist = RouteDistance;

  gstations = NumberofGroundStations;
  nodes = NumberofSatellites;
  MAXLINKS = MaxLinks;

  TempPacket = (SatDSwdpayload_t*)inbound_update.CopyArc();

  int h,i,g;
  int j,k; // loop counters
  simtime = TNow();

  i = (*TempPacket)->destination;
  h = (*TempPacket)->source;
  g = (*TempPacket)->sequencenumber;

  //Recover dtable from NodeMemory
  tempiv = (INTVECTOR_t&)nodemem[i];
  for (j=0;j<nodes;j++)
    for (k=0;k<MAXLINKS;k++)
      dtable[j][k] = tempiv[j*MAXLINKS+k];
    for (j=0;j<nodes;j++)
      for (k=0;k<MAXLINKS;k++)
        htable[j][k] = tempiv[nodes*MAXLINKS + j*MAXLINKS+k];
    for (k=0;k<MAXLINKS;k++)
      htable[i][k] = i;

  //Recover this node's working environment
  for (j=0;j<MAXLINKS;j++) neighbor[j] = NeighMatrix[i*MAXLINKS+j];

  int link =-1;
  for (k=0;k<MAXLINKS;k++) { //What link is h on?
    if (neighbor[k] == h) link=k;
    }//for k

  //Input changes to dtable -- (1) from Garcia's algorithm
  if (link > -1) { //Only do updates for recognized neighbors
    payload = (*TempPacket)->Payload;
    int dest, dist, d_ih;
    int length=payload.Length();
    d_ih = (int)SATDIST(i,h); //Assumes path to h is direct
    Vi.ChangeLength(1,-1);
```

```
for (j=0;j<length;j=j+3) {
  dest=payload[j];
  dist=payload[j+1];
  if (dest != -1 && dest != i) {
dtable[dest][link] = d_ih + dist;
htable[dest][link] = payload[j+2];
if (dtable[dest][link] > (int)MAXDIST)
  dtable[dest][link] = (int)MAXDIST;
}//if
 }//for j -- all pairs in input
 htable[h][link] = i; //Head of path to neighbor is current node.
}// if recognized neighbor

//Update routing table from dtable -- Garcia (2)
int b;
int flag = 0;
//Determine if the new data has changed our preferred neighbor
for (j=0;j<nodes;j++) {
    b = dtable_min(j);
if (b < 0 ) b=0; //If b=-1, there is no path to dest, use any link
if (dtable[j][b] != RDist[i*nodes+j] && j != i) {
  flag = -1;
  break;
  }
  }//for j

if (flag == -1) {
  rt_update(i);
} else {
  Vi.ChangeLength(1,-1);
}//if

//Send updates to neighbors -- Garcia (3)
int t;
INTVECTOR_t outvect; outvect.ChangeLength(nodes*3,-1);

if (Vi.Length() > 1) for (b=0;b<MAXLINKS;b++) {
  if (neighbor[b] != -1) {
   for (t=0;t<nodes;t++) {
    if (in_path(i,b,Vi[t*3])) {
  outvect[t*3+0] = Vi[t*3+0];    //node
  outvect[t*3+1] = (int)MAXDIST; //if in path send infinity
  outvect[t*3+2] = -1;           //and invalid head node
     } else {
  outvect[t*3+0] = Vi[t*3+0];    //node
  outvect[t*3+1] = Vi[t*3+1];    //distance
  outvect[t*3+2] = Vi[t*3+2];    //head node
 }//if in_path
}//for t -- all nodes
  (*TempPacket)->source = i;
  (*TempPacket)->packetlength = outvect.Length() * 32 + PACKETHEADERSIZE;
  (*TempPacket)->sequencenumber = seqnum++;
  (*TempPacket)->Payload = outvect;
  (*TempPacket)->destination = neighbor[b];
  OutList.Enqueue(*TempPacket);
  NextPacket_Entry().Schedule(0,trig);
  }//if neighbor[b] != -1
 }//for b -- all neighbors

//Re-pack node's environment back into the global variables
for (j=0;j<nodes;j++)
  for (k=0;k<MAXLINKS;k++)
    tempiv[j*MAXLINKS+k] = (int)dtable[j][k];
for (j=0;j<nodes;j++)
  for (k=0;k<MAXLINKS;k++)
    tempiv[nodes*MAXLINKS + j*MAXLINKS+k] = htable[j][k];
nodemem[i] = tempiv;
SetNodeMemory(nodemem);
for (j=0;j<MAXLINKS;j++) NeighMatrix[i*MAXLINKS+j] = neighbor[j];
```

```
    SetNeighbors(NeighMatrix);

    if (Vi.Length() > 1) {

     //Update the Satcom-style RouteMatrix (matrix of next-hops)
     for (j=0;j<nodes;j++) {
      if (i != j) {

     if (routes[j][0] != MAXLINKS+2 && routes[j][0] != -1) {
       RouteMatrix[i*nodes+j] = neighbor[routes[j][0]];
      } else {
       RouteMatrix[i*nodes+j] = -1;
      }//if

      } else {

        RouteMatrix[i*nodes+j]=i;
      }//if (i != j)

      RDist[i*nodes+j] = routes[j][1];
     } //for j

     SetRoutingtablememory(RouteMatrix);
     SetRouteDistance(RDist);

    }//if length Vi > 1

  //Output old-style Satcom matrix
  routematrix(RouteMatrix);

  delete(TempPacket);
}

//*********************************************************************
// Asynchronous Functions:
inline void exBF::NextPacket_Run(const TRIGGER_t& NextPacket)
{
  SatDSwdpayload_t *OutPacket;
  simtime = TNow();
  int i = OutList.Length();
  if (i>0) {
    OutPacket = (SatDSwdpayload_t *)OutList.Dequeue();
    outbound_update(*OutPacket);
    delete(OutPacket);
  }//if i>1
 }

/****  User Code Above Here ****/
```

B.3 Xref

Because the two routing protocols only route between satellites and SatLab generates the distances of groundstations and satellites in one table, it is necessary to provide a cross-referencing feature that tells the simulation which satellite is closest to a particular ground station. Rather than do this entirely in Designer primitives, the link formation subroutine from the routing algorithms can be easily adapted to do the job as a stand-alone custom primitive. The following "xref" primitive does just that.

o External Ports:

Input: Update is of type TRIGGER

o External Arguments:

(M) "TranslationVector" is of type "INT-VECTOR"

Description: Memory which stores the node number of the satellite closest to each groundstation.

(M) "Distance table memory" is of type "REAL-VECTOR"

(M) "Number of Satellites" is of type "INTEGER"

(M) "Number of GroundStations" is of type "INTEGER"

o Internal Arguments:

* None *

```
/*
 *        Module Name :                xref
 *        Template Created By :        3.0
 *        Author:                      rjanoso
 *        Last Modification Date:      1-Aug-1996 12:06:33
 *        Template Date:               1-Aug-1996 12:06:37
 */

/****  Includes and Defines Below Here ****/
#define MAXDIST 1000000000
/****  Includes and Defines Above Here ****/

/****  Instance Definitions Below Here ****/
  // void Init();
/****  Instance Definitions Above Here ****/

/****  User Constructor Code Below Here ****/
/****  User Constructor Code Above Here ****/

/****  User Code Below Here ****/
```

When triggered by a satellite position update cycle, this routine searches through the distance table memory for the closest satellite to each groundstation and enters the satellite's node number in the translation vector element corresponding to the groundstation node number. A -1 is entered if no satellite is in range.

```
// Run Functions:
inline void xref::Update_Run(const TRIGGER_t& Update)
{
  int a,g,s;
  int nodes = NumberofGroundStations + NumberofSatellites;
  INTVECTOR_t V = TranslationVector;
```

```
  if (V.Length() < NumberofGroundStations)
    V.ChangeLength(NumberofGroundStations,-1);

  for(g=0; g<NumberofGroundStations; g++) {

    //Find closest neighbor
    int best = -1;
    float bdist = MAXDIST;
    for (s=0; s<NumberofSatellites; s++) {
      a = Distancetablememory[g*nodes+(s+NumberofGroundStations)];
      if (a<bdist) {
    best = s;
    bdist = a;
    }//if
      }//for s -- best and bdist should now have closest sat

    V[g] = best;
  }//for g
  SetTranslationVector(V);
}

/****  User Code Above Here ****/
```

B.4 Add Element to Vector

The last custom primitive simply adds some functionality that seemed to be missing from the provided Designer run-time library. It takes an input vector, increases its length by one, and places the integer input into the new position.

o External Ports:

Input: InVect is of type INT-VECTOR

Output: OutVect is of type INT-VECTOR

Input: IntIn is of type INTEGER

o External Arguments:

* None *

o Internal Arguments:

* None *

```
/*
 *        Module Name :                Add Element to Vector
 *        Template Created By :        3.0
 *        Author:                      rjanoso
 *        Last Modification Date:      17-Aug-1996 17:46:50
 *        Template Date:               17-Aug-1996 17:46:54
 */

/****  Includes and Defines Below Here ****/
/****  Includes and Defines Above Here ****/

/****  Instance Definitions Below Here ****/
```

```
  // void Init();
/****  Instance Definitions Above Here ****/

/****  User Constructor Code Below Here ****/
/****  User Constructor Code Above Here ****/

/****  User Code Below Here ****/

// Run Function:
inline void AddElementtoVector::Run(const INTEGER_t& IntIn, const INTVECTOR_t& InVect)
{
int newlength = InVect.Length()+1;
INTVECTOR_t NewVect = InVect;
NewVect.ChangeLength(newlength);
NewVect[newlength-1] = IntIn;
OutVect(NewVect);
}

/****  User Code Above Here ****/
```

APPENDIX C

Modifications to the Darting Algorithm

Correct operation of the Darting algorithm assumes that traffic flows through all branches of the network; thus eventually disseminating compete topological information to all nodes. If this is not the case, it is possible for situations to arise where Darting will not converge to an optimal configuration, and predecessor update packets will be generated indefinitely. To alleviate this problem, a new type of "ping" packet was added to the algorithm to enable nodes that detect this type of discrepancy to exercise the portion of the network that is in question.

Let a subsection of the network be in the state shown in Figure 47, and let the local state at node **a** be as shown in Table 4. Let there be a steady stream of traffic from **a** to **g**. Further, let this traffic be traversing the optimal path a-c-d-h-g at a cost of 4.

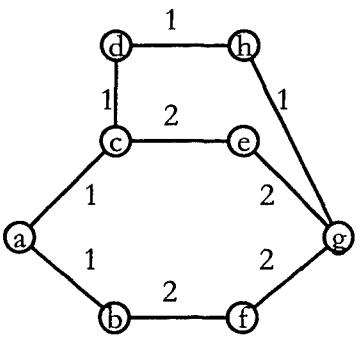

Figure 47: Degenerate Topology 1

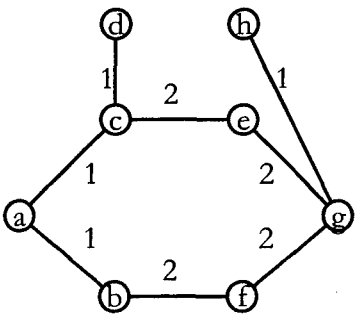

Figure 48: Degenerate Topology 2

Table 4: Initial State of Node A

Known links at a:
a->b=1
a->c=1
b->f=1
c->d=1
c->e=2
d->h=1
e->g=3
f->g=2
g->h=1

Now, let the link from **d** to **h** fail, as shown in Figure 48. When **a** next attempts to transmit to **g**, it will choose to go though **c** with an estimated cost from **c** to **g** of 3. For this

first packet, **c** will agree with the estimate of 3 and forward the packet on to node **d**. Node **d**, having detected the failure of the **d-h** link, will return the packet to node **c** along with the information about the failed link. Node **c** will then realize that its best path to **g** lies through **e** and will forward the packet accordingly. No further update packets are sent because the predecessor update mechanism only updates one node upstream from the point at which the discrepancy is detected.

Now for the next packet, **a** (not having any new information) will again choose to forward through node **c**. At this point, however, **c** will realize that **a** is using outdated information because **c**'s cost to **g** is now 4, not 3. Unfortunately, when **c** builds the update packet to send back to **a**, it enters the link information from the downstream nodes *as perceived by node c*. Thus **c** will update **a** with {c-d=1, c-e=2, e-g=2, g-h=1}. However, none of this is new information to node **a**! Thus, **a**'s behavior will not change, and it will continue to estimate the cost from **c** to **g** as 3 (via node **d**) and **c** will continue to try to update **a** with the correct cost of 4, via node **e**. This will go on indefinitely.

The problem occurs because in this instance, no traffic ever returns to node **a** through the path from node **d**. A mechanism is needed to force traffic to flow through that path to break the update cycle. Toward this end, a new type of "ping" packet was introduced into the protocol, and the contents of the fields in a predecessor update packet were slightly altered. The cost field was changed to reflect the cost to get from node **c** to node **g** (the cost **a** should have calculated), and the source field in a type 3 packet now has the value of "**g**" instead of "**c**".

Now, when node **a** receives the predecessor update packet from node **c**, it can check to see if the cost asserted by node **c** matches the cost it calculates from its local tables (after being updated with the new link information from **c**). If there is still a discrepancy, node **a** generates a ping packet addressed to node **d**. The ping packet is treated just like a data packet by each of the intermediate nodes, which place their local link data into the SCL field of the packet. Once the packet reaches node **d**, it is turned around by reversing the source

and destination fields, and returned to node **a**. Thus, the path from **d** to **a** is exercised, and **a** will receive the information about the failed link and adjust accordingly.

Bibliography

AdR87 W. S. Adams and L. Rider, "Circular polar constellations providing continuous single or multiple coverage above a specified latitude," The Journal of the Astronautical Sciences, 35: 155-192 (April-June 1987)

Alt94 Alta Group. Bones Designer Modeling Guide, Chapter 4, Alta Group of Cadence Design Systems, Foster City, CA, 1994.

AnL91 Ansari, N., and Liu, D. "The Performance Evaluation of a New Neural Network Based Traffic Scheme for a Satellite Communication Network," GLOBECOM 1991. 110-114. New York: IEEE Press, 1991.

ArA94 Arulambalam A., and Ansari, N. "Traffic Management of a Satellite Communication Network Using Mean Field Annealing," IEEE 1994 International Conference on Neural Networks. 3577-3582. New York: IEEE Press, 1994.

Ash90 Ash, G. "Design and Control of Networks with Dynamic Nonhirearchical Routing," IEEE Communications Magazine, 28: 34-40 (October 1990).

BaA93 Balasekar, S., and Ansari, N. "Adaptive Map Configuration and Dynamic Routing to Optimize the Performance of a Satellite Communication Network," GLOBECOM 1993. 986-990. New York: IEEE Press, 1993.

BaC96 Banks, J., Carson, J., Nelson, B. Discrete-Event System Simulation, 447-449. Prentice Hall, Upper Saddle River, NJ, 1996.

BiH87 Binder, R., Huffman, S., Gurantz, I., and Vena, P. "Crosslink Architectures for a Multiple Satellite System," Proceedings of the IEEE, 75: 74-81(Jan 87).

CaA87 Cain, J., Adams, S., Noakes, M., and Kryst, T. "A Near-Optimum Multiple Path Routing Algorithm for Space-Based SDI Networks," Vehicular Technology Conference 1994. 578-585. New York: IEEE Press, 1987.

Cha89 Chakraborty, D. "Survivable Communications Concept via Multiple Low-Earth-Orbiting Satellites," IEEE Transactions on Aerospace and Electronic Systems, 25: 881-889 (1989).

Che86 Cheriton, D. "VMTP A Transport Protocol For The Next Generation Of Communication Systems," SIGCOMM 1986. 406-415. New York: ACM Press, 1986.

ChR89 Cheng, C., Riley, R., Kumar, S., and Garcia-Luna-Aceves, J. "A Loop Free Extended Bellman-Ford Routing Protocol Without Bouncing Effect," SIGCOMM 1989. 224-236. New York: ACM Press, 1989.

ClJ89 Clark D., Jacobson V., Romkey J., and Salwen H., "An analysis of TCP processing overhead", IEEE Communications Magazine, 27: 23-29 (June 1989)

CoL90 Corman, T., Leiserson, C., and Rivest, R. Introduction to Algorithms, 527-529. MIT Press, Cambridge MA, 1990.

FCC91 Federal Communications Commission. "Comments of the Hughes Aircraft Company" In the Matter of the Application of Motorola Satellite Communications Inc.,, For a Low Earth Orbit Based Mobile Satellite Communications System. File Nos. 9-DSS-P-91(87) CSS-91-010, 3 June 1991.

Fol95 Foley, T. "WRC backs broadband satellite plan," Communications Week International, via World Wide Web, http://techweb.cmp.com/cwi, Nov 1995.

GaG94 Ganz, A., Gong, Y., and Li, B. "Performance Study Of Low Earth Orbit Satellite Systems," IEEE Transactions on Communications, 42: 1866-1871 (Feb-April 1994).

GrZ89 Gross, J and Ziemer, R. "Distributed Routing Network Performance In Hostile Environments," Proceedings of the SPIE, 1059: 22-26 (1989).

Jaf84 Jaffe, J. "Algorithms For Finding Paths With Multiple Constraints," Networks, 14: 95-116 (1984)

Kle75 Kleinrock, Leonard. Queueing Systems, Volume 1: Theory John Wiley & Sons, New York, 1975.

KoJ88 Kosowsky, R., Jacobs, L., and Gillhousen, K. "ARNS: A New Link Layer Protocol," MILCOM 88. 515-519. New York: IEEE Press, 1988.

KoK94 Kota, S., and Kallus, J. "Reservation Access Protocol for a Multiplanar ATM Switched Satellite Network," MILCOM 1994. 1048-1051. New York: IEEE Press, 1994.

KuS84 Kung, R., and Shacham, N. "An Algorithm For The Shortest Path Under Multiple Constraints," GLOBECOM 84. 355-359. New York: IEEE Press, 1984.

Leo91 Leopold, R. "Low Earth Orbital Global Cellular Communications Network," Proceedings of the ICC, 35: A.2.1 - A.2.4 (1991)

PrB86 T. Pratt and C. W. Bostian, Satellite Communications, John Wiley & Sons, New York, 1986.

PuP92 Pullman, M. A., Peterson, K.M. and Jan, Y. "Meeting The Challenges Of Applying Cellular Concepts To LEO SATCOM Systems," ICC 1992. 770-773. New York: IEEE Press, 1992.

Rou93 Rouffet, D. "GLOBALSTAR: a Transparent System," Electrical Communications: 84-90 (Q1, 1993)

Sat95 Alta Group. SatLab User's Guide, Chapter 7, Alta Group of Cadence Design Systems, Foster City, CA, 1995.

Sha88 Shacham, N. "Protocols For Multi-Satellite Networks," MILCOM 1988. 501-505. New York: IEEE Press, 1988.

ShA92 Shankar, A. Udaya, Alaettinoglu, Cengiz, Dussa-Zieger, Klaudia, and Matta, Ibrahim. "Performance Comparison of Routing Protocols under Dynamic and Static File Transfer Connections," SIGCOMM 1992. 39-52. New York: ACM Press, 1992.

Ste96 Stenger, D., *The Determination of the Minimum Acceptable Constellation for the Iridium Low Earth Orbit Satellite Network.* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH AFIT/GCS/ENG/96D, December 1996

TsC94 Tsai, Z., Chuang C., Cjang, J., and Huang, C. "Performance Of a Global Circuit Switched Satellite Communication Network," Vehicular Technology Conference 1994. 1624-1629. New York: IEEE Press, 1994.

TsM95 Tsai, K., and Ma, R. "Darting: A Cost Effective Routing Alternative For Large Space-Based Dynamic Topology Networks," MILCOM 1995. 682-687. New York: IEEE Press, 1995.

Tuc93 Tuck, E. "The Calling Network: a global telephone utility," Space Communications, 11: 141-161 (1993)

VoP93 Vojcic, B. R., Pickholtz, R. L., Milstein L. B. "Effects of Imperfect Power Control on a CDMA System Operating Over a Low Earth Orbit Satellite Link," MILCOM 1993. 973-977. New York: IEEE Press, 1993.

Wal77 J. G. Walker, "Continuous whole-earth coverage by circular-orbit satellite patterns," Royal Aircraft Establishment, Technical Report 77044, September 1977.

Wis95 Wisloff, T. "A tabulated overview of big LEOs," via World Wide Web, Tor.E.Wisloff@idt.unit.no, Oct 1995.

WuM94 Wu, W., Miller, E., Pritchard, W., and Pickholtz, R. "Mobile Satellite Communications," Proceedings of the IEEE, 82: 1431-1446 (Sept 1994)

YaG93 Yang, Wen-Bin and Geraniotis. "Performance Analysis Of Networks Of Low Earth Orbit Satellites With Integrated Voice/Data Traffic," MILCOM 1993. 978-982. New York: IEEE Press, 1993.

Vita

Richard F. Janoso [redacted] He graduated from high school in Allentown, Pennsylvania in 1986 and attended Lehigh University in Bethlehem, Pennsylvania, from which he received a Bachelor of Electrical Engineering degree in 1990. He received his United States Air Force commission from the Reserve Office Training Corps and entered active duty in February 1991. His first assignment was to the 485th Engineering Installation Group, Griffiss AFB, New York, where he served as a transmission systems engineer, local area network design engineer, and Chief, Cable Installation Branch. He entered the School of Engineering, Air Force Institute of Technology, in June 1995. He is a member of Tau Beta Pi and Eta Kappa Nu.

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1996	Master's Thesis

4. TITLE AND SUBTITLE
PERFORMANCE ANALYSIS OF DYNAMIC ROUTING PROTOCOLS IN A LOW EARTH ORBIT SATELLITE DATA NETWORK

5. FUNDING NUMBERS

6. AUTHOR(S)
Richard F. Janoso

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/96D-08

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. Joseph Liu
HQ SWC/AES
730 Irwin Ave, Suite 83
Falcon AFB, CO 80912-7383

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Modern warfare is placing an increasing reliance on global communications. Currently under development are several Low Earth Orbit (LEO) satellite systems that propose to deliver voice and data traffic to subscribers anywhere on the globe. However, very little is known about the performance of conventional routing protocols under orbital conditions where the topology changes in minutes rather than days. This briefing compare s two routing protocols in a LEO environment. One (Extended Bellman-Ford) is a conventional terrestrial routing protocol, while the other (Darting) is a new protocol which has been proposed as suitable for use in LEO networks. These protocols were compared via computer simulation in two of the proposed LEO systems (Globalstar and Iridium), under various traffic intensities. Comparative measures of packet delay, convergence speed, and protocol overhead were made.

14. SUBJECT TERMS	15. NUMBER OF PAGES
low earth orbit, routing, satellite, network simulation	117
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.